National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

DSP IMPLEMENTATION OF ADAPTIVE RF POWER AMPLIFIER LINEARIZERS

by

© **I-KEONG LAU**

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL ENGINEERING

EDMONTON, ALBERTA

FALL 1994

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-95066-8

Canadä

Name Lau, I-Keong

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Engineering, Electronics and Electrical
SUBJECT TERM

0544
SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture	0729
Art History	0377
Cinema	0900
Dance	0378
Fine Arts	0357
Information Science	0723
Journalism	0391
Library Science	0399
Mass Communications	0708
Music	0413
Speech Communication	0459
Theater	0465

EDUCATION

General	0515
Administration	0514
Adult and Continuing	0516
Agricultural	0517
Art	0273
Bilingual and Multicultural	0282
Business	0688
Community College	0275
Curriculum and Instruction	0727
Early Childhood	0518
Elementary	0524
Finance	0277
Guidance and Counseling	0519
Health	0680
Higher	0745
History of	0520
Home Economics	0278
Industrial	0521
Language and Literature	0279
Mathematics	0280
Music	0522
Philosophy of	0998
Physical	0523
Psychology	0525
Reading	0535
Religious	0527
Sciences	0714
Secondary	0533
Social Sciences	0534
Sociology of	0340
Special	0529
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language	
General	0679
Ancient	0289
Linguistics	0290
Modern	0291
Literature	
General	0401
Classical	0294
Comparative	0295
Medieval	0297
Modern	0298
African	0316
American	0591
Asian	0305
Canadian (English)	0352
Canadian (French)	0355
English	0593
Germanic	0311
Latin American	0312
Middle Eastern	0315
Romance	0313
Slavic and East European	0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy	0422
Religion	
General	0318
Biblical Studies	0321
Clergy	0319
History of	0320
Philosophy of	0322
Theology	0469

SOCIAL SCIENCES

American Studies	0323
Anthropology	
Archaeology	0324
Cultural	0326
Physical	0327
Business Administration	
General	0310
Accounting	0272
Banking	0770
Management	0454
Marketing	0338
Canadian Studies	0385
Economics	
General	0501
Agricultural	0503
Commerce-Business	0505
Finance	0508
History	0509
Labor	0510
Theory	0511
Folklore	0358
Geography	0366
Gerontology	0351
History	
General	0578
Ancient	0579
Medieval	0581
Modern	0582
Black	0328
African	0331
Asia, Australia and Oceania	0332
Canadian	0334
European	0335
Latin American	0336
Middle Eastern	0333
United States	0337
History of Science	0585
Law	0398
Political Science	
General	0615
International Law and Relations	0616
Public Administration	0617
Recreation	0814
Social Work	0452
Sociology	
General	0626
Criminology and Penology	0627
Demography	0938
Ethnic and Racial Studies	0631
Individual and Family Studies	0628
Industrial and Labor Relations	0629
Public and Social Welfare	0630
Social Structure and Development	0700
Theory and Methods	0344
Transportation	0709
Urban and Regional Planning	0999
Women's Studies	0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture	
General	0473
Agronomy	0285
Animal Culture and Nutrition	0475
Animal Pathology	0476
Food Science and Technology	0359
Forestry and Wildlife	0478
Plant Culture	0479
Plant Pathology	0480
Plant Physiology	0817
Range Management	0777
Wood Technology	0746
Biology	
General	0306
Anatomy	0287
Biostatistics	0308
Botany	0309
Cell	0379
Ecology	0329
Entomology	0353
Genetics	0369
Limnology	0793
Microbiology	0410
Molecular	0307
Neuroscience	0317
Oceanography	0416
Physiology	0433
Radiation	0821
Veterinary Science	0778
Zoology	0472
Biophysics	
General	0786
Medical	0760

EARTH SCIENCES

Biogeochemistry	0425
Geochemistry	0996
Geodesy	0370
Geology	0372
Geophysics	0373
Hydrology	0388
Mineralogy	0411
Paleobotany	0345
Paleoecology	0426
Paleontology	0418
Paleozoology	0985
Palynology	0427
Physical Geography	0368
Physical Oceanography	0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences	0768
Health Sciences	
General	0566
Audiology	0300
Chemotherapy	0992
Dentistry	0567
Education	0350
Hospital Management	0769
Human Development	0758
Immunology	0982
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and Therapy	0354
Ophthalmology	0381
Pathology	0571
Pharmacology	0419
Pharmacy	0572
Physical Therapy	0382
Public Health	0573
Radiology	0574
Recreation	0575
Speech Pathology	0460
Toxicology	0383
Home Economics	0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry	
General	0485
Agricultural	0749
Analytical	0486
Biochemistry	0487
Inorganic	0488
Nuclear	0738
Organic	0490
Pharmaceutical	0491
Physical	0494
Polymer	0495
Radiation	0754
Mathematics	0405
Physics	
General	0605
Acoustics	0986
Astronomy and Astrophysics	0606
Atmospheric Science	0608
Atomic	0748
Electronics and Electricity	0607
Elementary Particles and High Energy	0798
Fluid and Plasma	0759
Molecular	0609
Nuclear	0610
Optics	0752
Radiation	0756
Solid State	0611
Statistics	0463

Applied Sciences

Applied Mechanics	0346
Computer Science	0984
Engineering	
General	0537
Aerospace	0538
Agricultural	0539
Automotive	0540
Biomedical	0541
Chemical	0542
Civil	0543
Electronics and Electrical	0544
Heat and Thermodynamics	0348
Hydraulic	0545
Industrial	0546
Marine	0547
Materials Science	0794
Mechanical	0548
Metallurgy	0743
Mining	0551
Nuclear	0552
Packaging	0549
Petroleum	0765
Sanitary and Municipal	0554
System Science	0790
Geotechnology	0428
Operations Research	0796
Plastics Technology	0795
Textile Technology	0994

PSYCHOLOGY

General	0621
Behavioral	0384
Clinical	0622
Developmental	0620
Experimental	0623
Industrial	0624
Personality	0625
Physiological	0989
Psychobiology	0349
Psychometrics	0632
Social	0451

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: **I-KEONG LAU**
TITLE OF PROJECT REPORT: **DSP IMPLEMENTATION OF ADAPTIVE RF POWER AMPLIFIER LINEARIZERS**
DEGREE: **MASTER OF SCIENCE**
YEAR THIS DEGREE GRANTED: **FALL 1994**

Permission is hereby granted to THE UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the project report, and except as hereinbefore provided neither the project report nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

I-KEONG LAU

Date: Aug 30, 1994

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **DSP IMPLEMENTATION OF ADAPTIVE RF POWER AMPLIFIER LINEARIZERS** submitted by **I-KEONG LAU** in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** in Electrical Engineering.

Paul A. Goud

Dr. Paul A. Goud, Co-Supervisor

Colin G. Englefield.

Dr. Colin G. Englefield, Co-Supervisor

R. E. Rink

Dr. Raymond Rink, Internal Examiner

KW Fyfe

Dr. Kenneth Fyfe, External Examiner

Date: 26th August 1994.

DEDICATED TO

My parents (especially my father Kuan Lau), brothers, sisters, and friends for their love, support, and encouragement over the years.

ABSTRACT

The performance of an enhanced RF power amplifier linearization method is investigated experimentally. A linear modulation technique, namely π/4 DQPSK, has been adopted by interim standard IS-54 for the North American digital cellular system to efficiently use the limited signal spectrum. This linear modulation technique generates a fluctuating signal envelope that causes intermodulation products and spectral spreading after non-linear amplification. This modulation technique therefore requires a highly linear amplifier to keep the distortion within acceptable limits. The complex gain predistortion linearization method has demonstrated excellent performance for linearizing class AB or class B power amplifiers. However, in practice, this linearization method suffers from several implementation problems such as sensitivity to initial conditions, intensive computational requirements that limit the input signal bit rate, and low look-up-table to sampling point (L/S) ratio.

The enhanced linearization method is an extension of the complex gain predistortion technique, and is implemented on a digital signal processor (DSP). The objective of this method is to achieve a practicable linearizer with a high input data bit rate, a short look-up-table (LUT) update time, relatively high L/S ratio requiring a small amount of memory, and moderate computational power. Three complex baseband signals are sampled simultaneously; namely, the quadrature modulated input and feedback signals, and the predistorted signal. Using these sampled data, a LUT with new complex predistortion values is calculated in a time efficient manner.

This enhanced linearization method was implemented on a TMS320C30 DSP development platform with dual 16 bit ADC/DAC ports. Object oriented programing and nonpreemptive multitasking techniques have been adopted in the software implementation in order to reduce the time for algorithm development, verification and testing as well as to optimize the processing speed. As a result, the implemented linearization method achieves a performance of: maximum sampling rate of 151.2 ksample/s, maximum input data bit rates of 30.3 kb/s, LUT updating rates of 33 tables/s, and LUT convergence time of ~2 seconds. This linearization method reduces the out-of-band spectral distortion by up to 28 dB, 15 dB and 10 dB in the first, second and third adjacent channel, respectively.

ACKNOWLEDGMENT

I would like to express my sincere thanks to my supervisors, Dr. Paul A. Goud and Dr. Colin G. Englefield for their advice, encouragement, and help throughout the course of this work. I would also like to thank Dr. Andrew Wright for useful implementation discussions.

I am indebted to Telecommunication Research Laboratories and its sponsors for their financial and technical support. I would also like to extend to my thanks to the research staff and graduate students at TR labs for their help and support.

My appreciation is due to Dobby Lam, Edmond Lou, Teresa Ng, Tim Neufeld, Anit Lohtia, and Yvonne J. den Otter. I am also grateful to the Ng's family. I thank my family and friends for their support and encouragement throughout the completion of this project.

TABLE OF CONTENTS

CHAPTER **PAGE**

1. INTRODUCTION **1**

1.1 Digital Cellular Mobile Radio Systems 1

1.2 Power Amplifier Linearization 3

1.3 Thesis Outline 4

2. RF AMPLIFIER DISTORTION AND LINEARIZATION TECHNIQUES **6**

2.1 Structure of Digital Transmitter 6

2.1.1 Data Source 7

2.1.2 Data Compressor and Frame/Slot Format 7

2.1.3 π/4 DQPSK Encoder 7

2.1.4 Pulse Shaping Filter 8

2.1.5 Quadrature Modulator 9

2.1.6 RF Oscillator 10

2.1.7 RF Power Amplifier 10

2.2 RF Power Amplifier Distortion 11

2.2.1 AM-AM Distortion 11

2.2.2 AM-PM Distortion 13

2.3 Nonlinear RF Amplifier Linearization Techniques 14

2.3.1 Feedforward Correction 15

2.3.2 Linear Amplification Using Nonlinear Components (LINC) 16

2.3.3 Cartesian Coordinate Negative Feedback 17

2.3.4 Adaptive Predistortion 18

2.3.4.1 Direct Mapping Predistortion 19

2.3.4.2 Complex Gain Predistortion 21

2.3.4.2.1 Secant Method 23

2.3.4.2.2 Rotate and Scale (RASCAL) Method 24

2.3.4.2.3 Cubic Spline Interpolation Method 25
2.4 Summary of Nonlinear RF Amplifier Linearization Techniques 27
2.5 Impairments in the Quadrature Modulator and Demodulator 30

3. ENHANCED LINEARIZATION METHOD 35

3.1 Delay Compensator 36
3.2 $H(|V_m|^2)$ Estimator 39
3.3 Mapping Controller 41
3.4 Linear Interpolator and Weighted Averaging Filter 44
3.5 Summary of this Chapter 47

4. IMPLEMENTATION OF THE ENHANCED LINEARIZER 48

4.1 Data Source 48
4.2 π/4 DQPSK Encoder and Pulse Shaping Filter 51
4.2.1 Design and Implementation of the π/4 DQPSK Encoder 52
4.2.1.1 Software Implementation 55
4.2.1.2 Hardware Implementation 56
4.2.2 Design and Implementation of the Pulse Shaping Filter 57
4.2.2.1 Software Implementation 63
4.2.2.2 Hardware Implementation 65
4.3 Enhanced Linearizer 66
4.3.1 Real-Time Predistortion Unit 68
4.3.2 Off-Line Distortion Estimation Unit 71
4.3.3 Power Regulation Unit 81
4.3.4 Clipping Control Unit 83
4.3.5 Loop Selector 85
4.4 Software Architecture of the Enhanced Linearizer 85
4.5 Quadrature Modulator, Power Amplifier and Quadrature Demodulator 92

5. RESULTS 97

5.1 Sampling Rate and Related Parameters 97

5.2 Evaluation for RF Power Amplifier Linearity 98
5.3 Test Results for the π/4 DQPSK Encoder and Pulse Shaping Filter 99
5.4 Test Results for the Enhanced Linearizer 102

6. SUMMARY AND CONCLUSIONS 108

REFERENCES 113

APPENDIX A. MINIMUM MEAN SQUARE ERROR (MMSE) TIMING RECOVERY ALGORITHM. 117

APPENDIX B. CIRCUIT DIAGRAMS FOR THE π/4 DQPSK ENCODER AND PULSE SHAPING FILTER. 121

APPENDIX C. NORMALIZED AM-AM AND AM-PM CONVERSION FACTORS FOR THE CLASS B AMPLIFIER, AND OP-AMP DIODE CIRCUIT. 124

APPENDIX D. MEMORY MAP FOR THE LINEARIZER DSP PROGRAMS. 126

APPENDIX E. PROGRAMMING LISTS FOR THE π/4 DQPSK ENCODER AND PULSE SHAPING FILTER, AND THE ENHANCED LINEARIZER. 128

LIST OF FIGURES

FIGURE PAGE

1-1 Maximum out-of-band transmitted power level in the IS-54 standard............. 2

2-1 Simplified block diagram of an IS-54 based digital transmitter. 6

2-2 Phase plane mapping and π/4 DQPSK constellation....................................... 8

2-3 Frequency response of square root raised cosine filter with 0.35 roll-off factor. 9

2-4 Block diagram of quadrature modulator. 10

2-5 Normalized input-output amplitude transfer characteristics of three different amplifiers.. 12

2-6 Normalized input-output phase transfer characteristics of three different amplifiers. 13

2-7 Block diagram of a digital mobile radio transmitter with linearization............ 15

2-8 Block diagram of feedforward linearization. 15

2-9 Block diagram of LINC linearization technique. 16

2-10 Block diagram of CALLUM linearization technique. 17

2-11 Block diagram of Cartesian Coordinate Negative Feedback. 18

2-12 Block diagram of adaptive predistortion technique....................................... 18

2-13 Model of a nonlinear power amplifier in direct mapping predistortion. 19

2-14 Block diagram of direct mapping predistortion. 20

2-15 Block diagram of complex gain predistortion... 22

2-16 Block diagram of complex gain predistortion using RASCAL method. 25

2-17 Block diagram of complex gain predistortion using cubic spline interpolation method. 27

2-18 Effects of the quadrature modulator and demodulator impairments (DC offset, gain imbalance and phase imbalance) on sinusoidal I and Q input signals. 31

2-19 Block diagram of quadrature modulator and demodulator impairment compensator. 33

3-1 Block diagram of the enhanced linearization method. 35

3-2 Relationship between $|V_m|^2$ and sampling intervals, i. 43

4-1 Block diagram of a simplified digital transmitter with the enhanced linearizer. 48

4-2 Diagram of the implemented data source. 50

4-3 Structural diagram of the π/4 DQPSK encoder and pulse shaping filter. 52

4-4 (a) Constellation state diagram. (b) $\Delta\phi s(b1k,b0k)$ state diagram. 54

4-5 Block diagram of the hardware implementation of the π/4 DQPSK encoder 56

4-6 Block diagram of a standard FIR filter. 59

4-7 Example of obtaining h_i at the sampling rate (f_s) and $N=25$. 60

4-8 Example of the output from the interpolating filter. 61

4-9 Representation of the error vector of the pulse shaped output. 62

4-10 Software architecture of the entire implemented DSP program 65

4-11 Block diagram of the interfacing circuit for the digital pulse shaping filter. 66

4-12 Simplified block diagram of the entire enhanced linearizer. 67

4-13 Detailed functional block diagram of the enhanced linearizer. 69

4-14 Out-of-band power spectral density for different sizes of LUT. 70

4-15 Characteristics of the anti-aliasing filter and the reconstruction filter in the ADC/DAC. 72

4-16 Operation of the discrete time delay compensator. 73

4-17 Block diagram of the delay calibration process. 75

4-18 Simulated pdf obtained from 30 simulation outputs with 1.25×105 symbols/trial. 76

4-19 Implemented structure of the mapping controller. 77

4-20 Structure of the circular buffer. 79

4-21 Software architecture of the enhanced linearizer. 87

4-22 Sample execution trace of the implemented linearizer program. 89

4-23 Structural diagram of the communication between the C30 DSP and the 486 PC. 90

4-24 Hierarchical diagram of the implemented C++ program objects. 91

4-25 Block diagram of the digitally emulated power amplifier. 93

4-26 Complex transfer characteristic of the emulated class B power amplifier. 94

4-27 Gain compression and phase characteristics of the emulated class B power amplifier. 94

4-28 Amplitude transfer characteristic of the op-amp diode circuit. 96

5-1 Test setup for the π/4 DQPSK encoder and pulse shaping filter. 100

5-2 Output spectrum of the π/4 DQPSK encoder and pulse shaping filter at a bit rate of 18.9 kb/s (Test 1). 101

5-3 Output spectrum of the π/4 DQPSK encoder and pulse shaping filter at a bit rate of 30.4 kb/s (Test 2). 101

5-4 Test setup for the digitally emulated power amplifier. 102

5-5 Test setup for the op-amp diode circuit. 103

5-6 Output spectrum of the enhanced linearizer using the emulated power amplifier at a bit rate of 18.9 kb/s (Test 1). 104

5-7 Output spectrum of the enhanced linearizer using the op-amp diode circuit at a bit rate of 18.9 kb/s (Test 2). 104

5-8 Output spectrum of the enhanced linearizer using the emulated power amplifier at a bit rate of 30.4 kb/s (Test 3). 105

5-9 Output spectrum of the enhanced linearizer using the op-amp diode circuit at a bit rate of 30.4 kb/s (Test 4). 105

6-1 Block diagram of a multichannel transmitter. 112

LIST OF TABLES

TABLE PAGE

1-1 SUMMARIZED SPECIFICATION OF THE IS-54 STANDARD 2

2-1 SUMMARY OF THE PERFORMANCE CHARACTERISTICS OF SEVERAL LINEARIZATION TECHNIQUES 29

2-2 SPECIFICATION OF THE QUADRATURE MODULATOR AND DEMODULATOR 33

4-1 RELATION BETWEEN THE SERIAL INPUT DATA BITS AND $\Delta\phi$. 52

4-2 RELATION BETWEEN THE SERIAL INPUT DATA BITS AND THE $\Delta\phi s(b1k,b0k)$ FUNCTION. 55

4-3 ARRANGEMENT BETWEEN PROGRAM OBJECTS AND PROGRAMS. 87

5-1 VALUE OF THE INPUT BIT RATE AND TRANSMITTED SIGNAL BANDWIDTH WHEN M IS 10 OR 16, AND THE SAMPLING RATE IS 151.2 KSAMPLE/S 98

5-2 RELATIONSHIP BETWEEN THE TESTING INPUT DATA BIT RATE AND THE CORRESPONDING CHANNEL SPACING 99

5-3 SPECIFICATIONS OF THE TESTS FOR THE $\pi/4$ DQPSK ENCODER AND THE PULSE SHAPING FILTER. 100

5-4 SPECIFICATIONS OF FOUR DIFFERENT TESTS FOR THE ENHANCED LINEARIZER 103

LIST OF ABBREVIATIONS

$\pi/4$ DQPSK	$\pi/4$ Shifted Differential Phase Shift Keying
AM-AM	Amplitude to Amplitude
AM-PM	Amplitude to Phase
CALLUM	Combined Analog Locked Loop Universal Modulator
cdf	Cumulative Distribution Function
CDMA	Code Division Multiple Access
DSP	Digital Signal Processor
EIA/TIA	Electronic Industries Association/Telecommunication Industry Association
FIR	Finite Impulse Response
IIR	Infinite Impulse Response
kbps	kilo bits per second
L/S	the ratio of the number of points in the LUT to the number of sampling points.
LINC	Linear Amplification Using Nonlinear Components
LUT	Look-Up-Table
OOP	Object Oriented Programming
pdf	Probability Distribution Function
QPSK	Quadrature Phase Shift Keying
RASCAL	RotAte and SCALe
ROM	Read-Only-Memory
TDMA	Time Division Multiple Access
T_{LUT}	Predistortion LUT Update Cycle
tps	tables per second
VCO	Voltage Controlled Oscillator
VSELP	Vector Sum Excited Linear Prediction
word	A basic memory unit in the TMS320C30 DSP. Each word has 32 bit memory.

LIST OF SYMBOLS

η	Power Conversion Efficiency
$A_m^{\ 2}$	Magnitude squared of $\mathbf{V}_m$
I	In-phase signal
I_a, Q_a	Demodulated feedback signal from the amplifier (I and Q form)
I_f, Q_f	Demodulated feedback signal from the amplifier (I and Q form)
I_m, Q_m	Original quadrature modulated signal (I and Q form)
I_p, Q_p	Predistorted quadrature signal (I and Q form)
I_p, Q_p	Predistorted quadrature signal (I and Q form)
Q	Quadrature- phase signal
$\mathbf{V}_{adj}$	Adjusting vector
$\mathbf{V}_{err}$	Error vector
$V_{err_{rms}}$	RMS error vector magnitude
$\mathbf{V}_f$	Demodulated feedback signal from the amplifier (vector form)
$\mathbf{V}_m$	Original quadrature modulated signal (vector form)
$\mathbf{V}_{\mathrm{measured}}$	The measured output vector
$\mathbf{V}_p$	Predistorted quadrature signal (vector form)
$\mathbf{V}_{ref}$	reference pulse shaped output vector

1. INTRODUCTION

Spectrally-efficient linear modulation techniques, such as $\pi/4$ DQPSK, require highly linear RF power amplification at the transmitter. The high linearity is needed so that intermodulation distortion products and spectral spreading of the transmitted signal will be kept below acceptable levels. At the same time, high RF amplifier efficiency is desired in order to minimize the d.c. power required. Satisfying the efficiency requirement is at the expense of increasing the amplitude and phase nonlinearity of the RF amplifier. Amplifier linearization is therefore necessary. Adaptive complex gain predistortion is an attractive digital signal processing technique for linearizing a power amplifier. However, this technique suffers from some practical problems, such as sensitivity to initial conditions and an intensive computational requirement that limits the input signal bit rate. Therefore, a more realizable linearization method for cellular mobile radio applications needs to be investigated.

1.1 Digital Cellular Mobile Radio Systems

To increase the capacity of mobile communications, new systems are being developed. These new systems use digital technology, such as efficient speech coding, error correcting channel codes, bandwidth efficient modulation, and time division multiple access (TDMA) or code division multiple access (CDMA), in order to more effectively use the limited spectrum, and increase mobile network capacity. Interim standard IS-54 [1] of the Electronic Industries Association/Telecommunication Industry Association (EIA/TIA) subcommittee TR45.3 on Digital Cellular systems, has been adopted for use in North America. This new standard accommodates dual mode operation, so is compatible with the previously adopted analog advanced mobile phone service (AMPS) system. It thus provides a smooth transition from the analog to the digitally implemented system. Table 1-1 and Figure 1-1 show a summarized specification and graphical representation of the maximum out-of-band transmitted power level of the IS-54 standard, respectively.

Table 1-1. SUMMARIZED SPECIFICATION OF THE IS-54 STANDARD.

RF Frequency Band (Dependent on the system)	835.02-844.98 Mhz (Mobile) 880.02-889.98 Mhz (Base)
Channel Bandwidth	30 kHz
Bit Rate	48.6 kb/s
Spectral Efficiency	1.62 b/s/Hz
Access Method	TDMA
Mobile RF Power	0.6-4 W
Out-of-Band Power for Digital Transmitter	-26 to -60dB (see Figure 1-1)
Modulation	$\pi/4$ DQPSK
Pulse shaping filter	Squared-Root-Raised-Cosine Filter ($\alpha = 0.35$)
Voice Compression Algorithm	VSELP
Demodulation	Differential or Coherent
Equalization	Adaptive

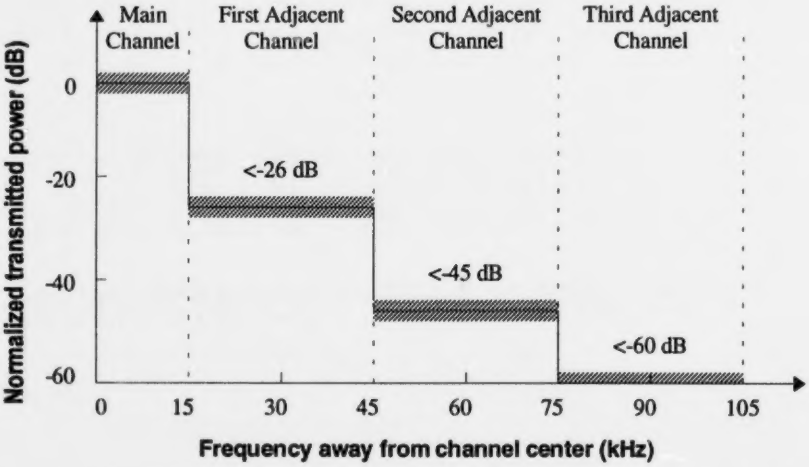


Figure 1-1. Maximum out-of-band transmitted power level in the IS-54 standard.

The IS-54 standard employs a linear digital modulation technique, called $\pi/4$ shifted differential quadrature phase shift keying ($\pi/4$ DQPSK), to reduce the input digital data bit rate by a factor of two, thus increasing the bandwidth efficiency. In the IS-54 standard, the transmission data rate of 48.6 kb/s and a channel spacing of 30 kHz results in a spectral efficiency of 1.62 b/s/Hz.

However, $\pi/4$ DQPSK modulation produces a modulated signal with a relatively large envelope variation, i.e., a 20 dB envelope variation. When the modulated signal is amplified by a nonlinear RF power amplifier which has a high power conversion efficiency, intermodulation products are present in the output of the amplifier, resulting in spectrum broadening and adjacent channel interference. The distortion in the output signal can be suppressed by using a linear power amplifier to amplify the modulated signal. However, the trade-off is that a low power conversion efficiency is achieved. Power amplifier linearization is used to obtain the good spectral characteristics of a linear amplifier, while at the same time obtaining the high power conversion efficiency that is usually associated with nonlinear amplifiers (i.e., class B or class C power amplifiers).

1.2 Power Amplifier Linearization

Several power amplifier linearization techniques, such as feedforward correction, linear amplification using nonlinear components, cartesian coordinate negative feedback, and Adaptive predistortion, have been proposed and studied. Adaptive complex gain predistortion is a digital linearization technique that can be applied to nonlinear RF power amplifiers. This technique assumes that the nonlinearity of the power amplifier can be modeled by memoryless distortion mechanisms; i.e., the amplitude and phase distortion of the amplifier are assumed to be dependent only on the input signal amplitude. The actual amplifier input signal is appropriately predistorted so that the output signal will be an undistorted amplified version of the original signal.

In adaptive complex gain predistortion, a base-band predistortion signal is generated digitally by multiplying the original base-band modulated signal by the corresponding predistortion vector before up converting it to the RF input signal of the nonlinear amplifier. Therefore, it is possible to implement this technique completely using a digital signal processor (DSP). However, since this technique requires intensive computation in generating the predistortion coefficients, an enhanced linearization method is studied in this project in order to achieve a practical linearization circuit with more moderate computational requirements, and a shorter calculation time for generating the predistortion coefficients.

1.3 Thesis Outline

In this chapter, relevant aspects of the digital cellular standard IS-54 have been discussed. Chapter 2 presents the basic structure of a digital transmitter which is based on the IS-54 standard. The causes and effects of RF power amplifier distortion when using filtered $\pi/4$ DQPSK modulation are described next. An overview of RF power amplifier linearization techniques, as well as of quadrature modulator and demodulator impairments is presented. Compensation of the quadrature modulator and demodulator impairments is also described.

Chapter 3 explains the structure of the enhanced linearization method which is studied in this thesis. The characteristic of each component in the enhanced method is also defined. The experimental implementation of a digital transmitter with the enhanced linearization method is described in Chapter 4.

Chapter 5 presents and discusses the test results obtained from the implemented linearizer. Finally, Chapter 6 summarizes the research results, and gives suggestions for future research work on this topic.

2. RF AMPLIFIER DISTORTION AND LINEARIZATION TECHNIQUES

The basic structure of a digital transmitter is presented in Section 2.1. The two components of RF power amplifier distortion, namely amplitude and phase distortion, are then described in Section 2.2. Background information on linearizing a nonlinear amplifier is presented next in Section 2.3 on nonlinear RF amplifier linearization techniques. Finally, the effects of impairments in the quadrature modulator and demodulator on the performance of RF amplifier linearizers are examined.

2.1 Structure of Digital Transmitter

A simplified block diagram for a digital transmitter, based on the IS-54 standard, is shown in Figure 2-1. This simplified digital transmitter contains seven parts: a data source, data compression and frame/slot format, π/4 DQPSK encoder, pulse shaping filter, quadrature modulator, local oscillator and RF power amplifier [2, 3]. The following subsections provide a brief description of these parts.

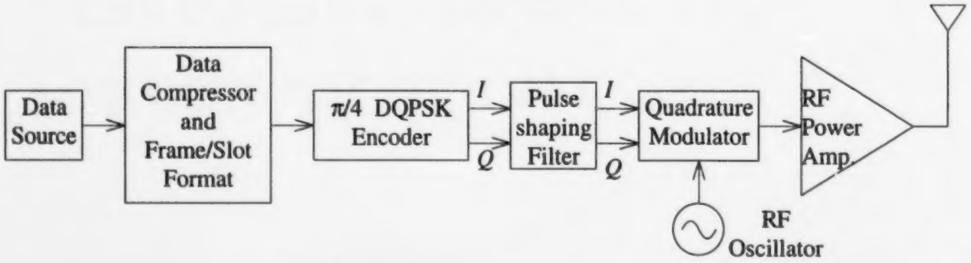

Data Source
Data Compressor and Frame/Slot Format
π/4 DQPSK Encoder
I
Q
Pulse shaping Filter
I
Q
Quadrature Modulator
RF Power Amp.
RF Oscillator

Figure 2-1. Simplified block diagram of an IS-54 based digital transmitter.

2.1.1 Data Source

The data source is the original digital data, such as a digitized analog signal, computer data or any form of digital data.

2.1.2 Data Compressor and Frame/Slot Format

For a digitized voice data source, a data compressor is used to reduce the data rate (i.e., the number of bits to be transmitted). According to the IS-54 standard, a linear predictive coding technique (VSELP) is used to compress the input digitized voice signal from 64 kbps to 13.2 kbps [1]. After compression, the output data are mapped onto a standard frame format, to produce a 48.6 kbps serial data stream.

2.1.3 $\pi/4$ DQPSK Encoder

In the $\pi/4$ DQPSK encoder, two serial input data bits are encoded to each transmitted symbol according to the Gray code format. As shown in Figure 2-2a, the Gray code format changes only one bit at a time with respect to the two adjacent states. This reduces the error rate by ensuring that the most likely symbol error will produce only one bit error. Since every two data bits are encoded to an adjacent transmitted symbol, the symbol rate is one half of the input data rate. Each symbol is encoded to a pair of in-phase and quadrature phase (I and Q) signals in accordance with the $\pi/4$ DQPSK modulation scheme. Figure 2-2b shows the signal constellation of $\pi/4$ DQPSK modulation. This modulation is obtained by superimposing two quadrature phase shift keying (QPSK) signal constellations, offset by $\pi/4$ radians, to form an eight-point signal constellation. As mentioned in Section 1.1, $\pi/4$ DQPSK modulation can achieve a spectral efficiency of

1.62 bits/Hz. However, an envelope variation of 20 dB (10:1 in amplitude) is also produced in the filtered modulated signal. It is because of this relatively large envelope variation that transmitter linearization is necessary.

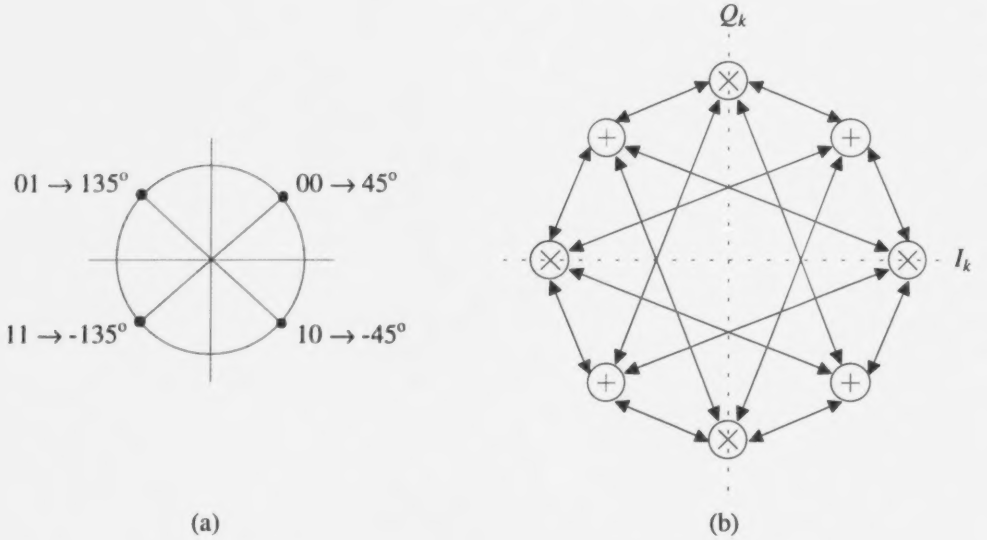


Figure 2-2. Phase plane mapping and π/4 DQPSK constellation.

2.1.4 Pulse Shaping Filter

The pulse shaping filter defines the spectral shape of the transmitted signal [1, 2]. According to the IS-54 standard, a square root raised cosine filter with a roll-off factor $\alpha = 0.35$ is used. Figure 2-3 shows the frequency response of this filter.

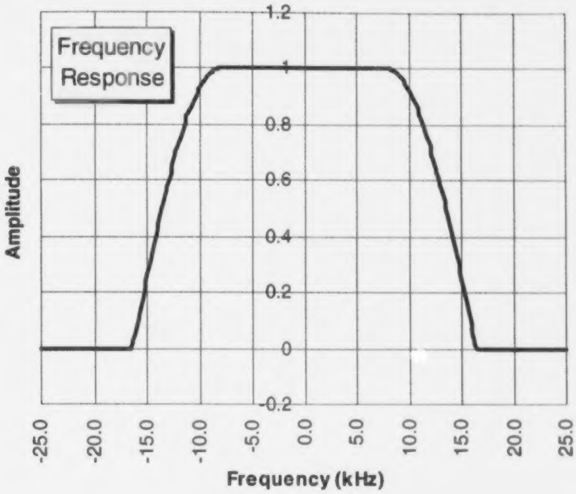


Figure 2-3. Frequency response of square root raised cosine filter with 0.35 roll-off factor.

2.1.5 Quadrature Modulator

The quadrature modulator consists of three components: a 90° phase splitter, two mixers, and an in-phase combiner. Their function is to combine the filtered quadrature signals (I and Q) and to produce the final transmitted RF signal. The 90° phase splitter generates two outputs with a 90° phase shift (i.e., $\sin(\omega t)$ and $\cos(\omega t)$) from the RF oscillator input. Two mixers correspondingly multiply these phase-shifted signals by the I and Q signals obtained from the $\pi/4$ DQPSK encoder with the pulse shaping filter. The outputs of the mixers are then added together by the in-phase combiner, resulting in the final $\pi/4$ DQPSK modulated RF signal. Figure 2-4 shows a block diagram of the quadrature modulator.

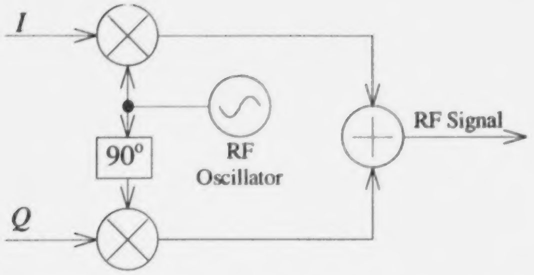


Figure 2-4. Block diagram of quadrature modulator.

2.1.6 RF Oscillator

The RF oscillator generates a stable carrier frequency, in one of the frequency ranges 835.02-844.98 MHz and 880.02-889.98 MHz. This is applied to the quadrature modulator in order to convert the baseband or intermediate frequency (IF) signal to a modulated RF signal.

2.1.7 RF Power Amplifier

Apart from the antenna, the RF power amplifier is the final stage of the digital transmitter, shown in Figure 2-1, and it is this part of the circuit that normally consumes the most electrical power. For linear modulation methods, a linear RF power amplifier must be used in order to maintain optimum spectral efficiency and low out-of-band emission. Such an amplifier however does not usually have a good d.c. to RF power conversion efficiency, i.e., a class A power amplifier has a maximum theoretical power conversion efficiency of ~50%. This is a major drawback for mobile applications, especially for the portable battery-operated unit, where battery life is of major importance. Higher power conversion efficiency can be obtained by using nonlinear power amplifiers, i.e., a class B or class C power amplifier. However, these amplifier distort the input signal, and cause spectral broadening and high out-of-band power emission of the output signal. In this case, it is necessary to apply amplifier linearization to the nonlinear amplifier in order to obtain the good spectral characteristics of a linear amplifier, as well as high power conversion efficiency.

2.2 RF Power Amplifier Distortion

In an ideal linear power amplifier, the output signal is linearly related to the input signal; i.e., the amplifier replicates the input signal with a constant gain and a constant time delay. However, because of power supply and gain mechanism limitations, a practical power amplifier provides linear amplification over only a limited operating range. Distortion is introduced into the output of the amplifier, especially when the amplifier is at or near to saturation, i.e., its maximum power level. These distortions vary the output signal amplitude and/or the phase of the amplifier, depending on the input signal level. Amplitude to amplitude (AM-AM) distortion and amplitude to phase (AM-PM) distortion are the two major distortion mechanisms in a nonlinear RF power amplifier.

2.2.1 AM-AM Distortion

AM-AM distortion causes variations in the output amplitude or RF envelope of a power amplifier. These variations arise from the nonlinear transfer characteristics of the power amplifier, and are dependent on the input signal level. Figure 2-5 shows the normalized transfer characteristics of three different amplifiers: an ideal amplifier, a class AB amplifier, and a class B amplifier.

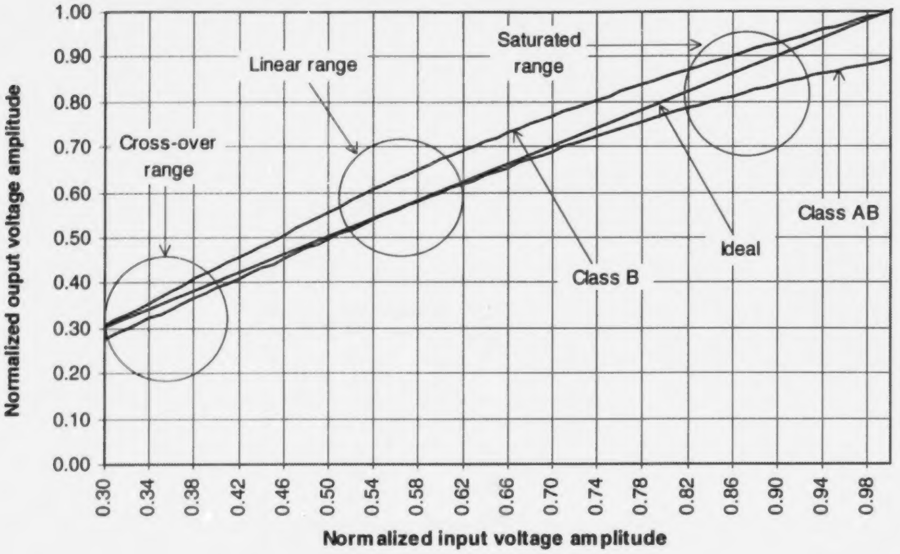


Figure 2-5. Normalized input-output amplitude transfer characteristics of three different amplifiers.

As shown in Figure 2-5, the ideal amplifier has a perfectly linear transfer characteristic; that is, the amplifier has a constant gain throughout the whole input signal level range. However, in the class AB and class B amplifier, the transfer characteristic is not a straight line, and can roughly be divided into three operational ranges: a cross-over range, a linear range, and a saturated range. In the cross-over range, the output of the amplifier, especially in the class B amplifier, is distorted due to the low input signal level and the proximity of the input threshold voltage of the amplifier. The distortion occurring in this range is called cross-over distortion. In the linear range, the amplifier operates in its active mode and has a nearly constant gain. However, in this operating range, the amplifier has a low DC-to-AC conversion efficiency. When the input signal level is increased beyond the linear range, the slope of the transfer characteristic decreases, and gain compression occurs. As a result of this non-linearity, the amplifier generates harmonic distortion and intermodulation distortion products. However, this is the region of high conversion efficiency.

2.2.2 AM-PM Distortion

AM-PM distortion in the output of an amplifier is caused by a non-constant signal delay, which is a function of the input signal level. Figure 2-6 shows typical phase variation characteristics of class AB and class B amplifiers.

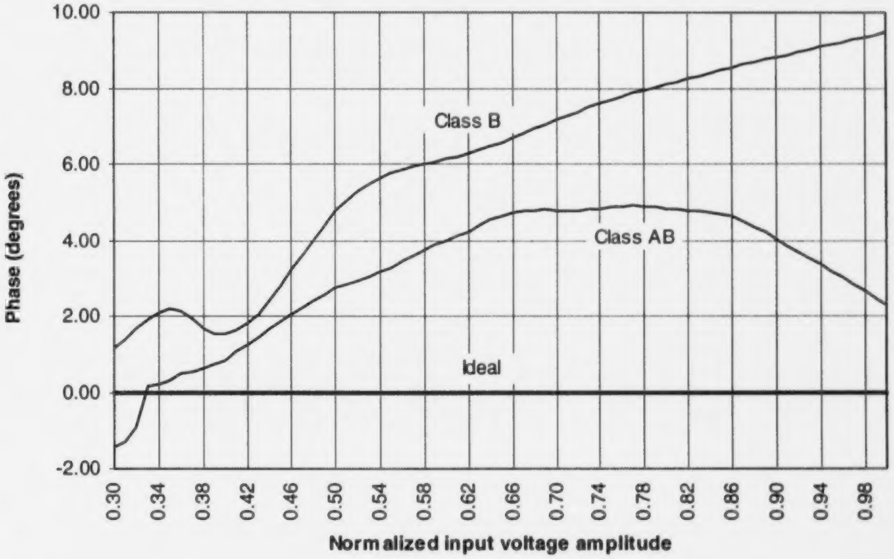


Figure 2-6. Normalized input-output phase transfer characteristics of three different amplifiers.

These phase variations in the amplifier generate harmonic distortion and intermodulation distortion products. Therefore, AM-PM distortion degrades the performance of a transmitter when the transmitted signal is a linearly modulated signal (e.g., filtered PSK modulation, DQPSK modulation, etc.).

2.3 Nonlinear RF Amplifier Linearization Techniques

Standard linear power amplifiers have poor power conversion efficiency and a limited dynamic range. The maximum theoretical power conversion efficiency (η) of a class A power amplifier is 50%. The maximum η of a class B amplifier is 75%. The maximum η of a class C amplifier is still higher, but at the cost of an even worse linearity. For a nonlinear power amplifier, therefore, η is approximately equal to or may be even greater than 75%. AM-AM distortion and AM-PM distortion, and the spectral broadening resulting from this nonlinearity are major disadvantages when a linear modulation method (e.g. $\pi/4$ DQPSK) is used to obtain high spectral efficiency. To minimize these problems, linearization techniques are used in the transmitter design in order to realize a linear amplifier with good power efficiency.

Figure 2-7 shows a general block diagram of a linearized digital mobile radio transmitter. A linearization circuit is added at the output of either the pulse shaping filter or the quadrature modulator in order to linearize the nonlinear RF power amplifier at baseband or at RF, respectively. Several linearization techniques have been proposed and studied. These techniques can be classified into the following four groups:

- ☐ Feedforward correction [3, 4, 5]
- ☐ Linear amplification using nonlinear components (LINC) [3, 4, 6, 7]
- ☐ Cartesian coordinate negative feedback [3, 4, 6, 8, 9, 10, 11]
- ☐ Adaptive predistortion [3, 6, 12, 13, 14]

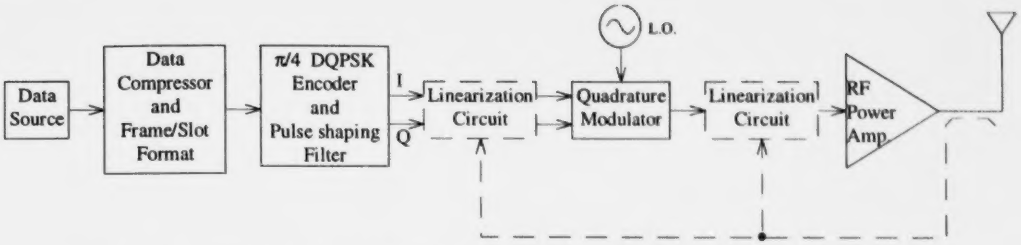

Data Source
Data Compressor and Frame/Slot Format
π/4 DQPSK Encoder and Pulse shaping Filter
I
Q
Linearization Circuit
L.O.
Quadrature Modulator
Linearization Circuit
RF Power Amp

Figure 2-7. Block diagram of a digital mobile radio transmitter with linearization.

2.3.1 Feedforward Correction

This linearization technique operates at RF, and uses two amplifiers to produce a linearly amplified signal. Figure 2-8 shows a block diagram of this technique. The main amplifier is a high-level nonlinear power amplifier, which is used to amplify the input signal. An error signal is generated by comparing an appropriately attenuated sample of the distorted main amplifier output signal with the delayed undistorted input signal. This error signal is then amplified by the low level auxiliary amplifier, which is a linear amplifier. The final output signal is obtained by subtracting the appropriately scaled auxiliary amplifier output from the delayed main amplifier output. Because this is an open system, it is unconditionally stable. However, changes in gain of the auxiliary amplifier due to aging and temperature changes are not compensated for by this technique.

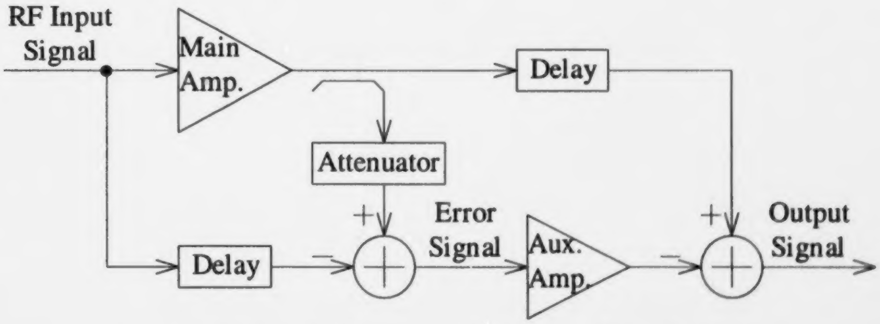

RF Input Signal
Main Amp.
Delay
Attenuator
+
Error Signal
Aux. Amp.
+
Output Signal
Delay
−
−

Figure 2-8. Block diagram of feedforward linearization.

2.3.2 Linear Amplification Using Nonlinear Components (LINC)

LINC, which is similar to feedforward correction, was developed in the 1930's. Figure 2-9 shows a block diagram of this linearization technique. LINC is an open loop system which separates the linearly modulated input signal into two anti-phase, constant envelope signals and then amplifies each of them using two well-matched nonlinear amplifiers. Since the amplified outputs contain anti-phase out-of-band components, summing these outputs will generate a linearly amplified signal. The difficulties in using this technique are mainly the matching of the amplifier characteristics, and generation of the constant amplitude signals.

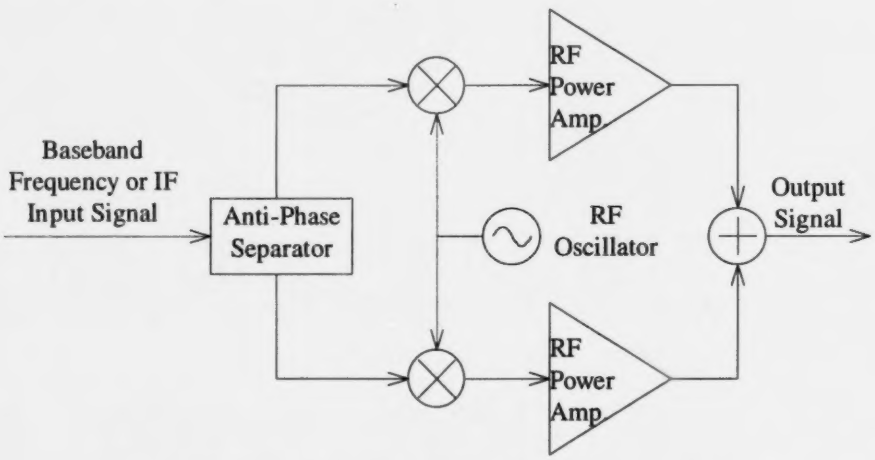


Figure 2-9. Block diagram of LINC linearization technique.

A modified technique, shown in Figure 2-10, is called combined analog locked loop universal modulator (CALLUM) [4, 15]. This technique adds a negative feedback loop and a voltage controlled oscillator (VCO) at each amplifier input in order to control the amplifier matching and to replace the anti-phase separator, respectively. Using this modified technique, a more complicated control circuitry is required. In addition, modulation format dependence, loop gain control ,and bandwidth restrictions degrade the amplification performance.

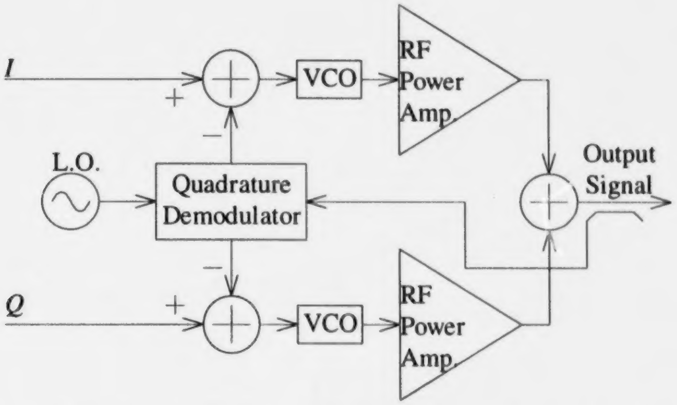


Figure 2-10. Block diagram of CALLUM linearization technique.

2.3.3 Cartesian Coordinate Negative Feedback

This technique, shown in Figure 2-11, applies a baseband negative feedback loop to linearize the RF power amplifier. The quadrature demodulated feedback signal $\mathbf{V}_f$, obtained by demodulating the power amplifier output, is subtracted from the quadrature input signal $\mathbf{V}_m$. After loop filtering, the quadrature loop error signal $\mathbf{V}_{err}$ is applied to the quadrature modulator and the power amplifier to form a closed feedback loop. The loop stability of this technique is sensitive to the loop gain and the phase shift. In order to keep the loop stable, good design of the loop gain and phase adjustment circuits is very important; otherwise, the loop may be unstable and oscillate. Due to the frequency-dependent delay of the negative feedback loop, the operating bandwidth of this technique is restricted. Cartesian coordinate negative feedback is a good narrow band technique for linearizing a power amplifier, and has the advantage that the circuitry is relatively straight forward. Several authors [9, 10, 11] have described the detailed analysis and implementation of this linearization technique.

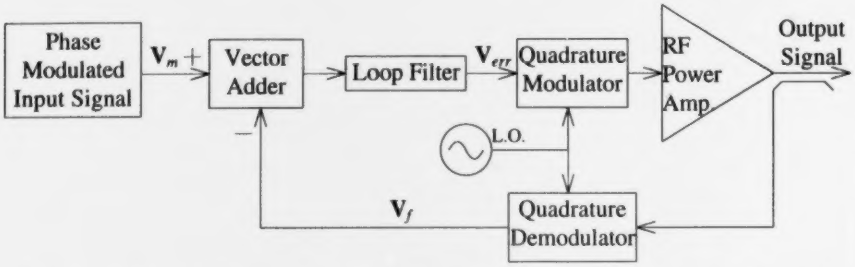

Phase Modulated Input Signal
V_m
+
Vector Adder
−
Loop Filter
V_{err}
Quadrature Modulator
RF Power Amp
Output Signal
L.O.
V_f
Quadrature Demodulator

Figure 2-11. Block diagram of Cartesian coordinate negative feedback.

2.3.4 Adaptive Predistortion

Adaptive predistortion was first described by Saleh and Salz in 1982 [12]. Using digital signal processing, this technique predistorts the baseband quadrature input signal to compensate for the nonlinear characteristics of the power amplifier. Therefore, the overall response of the power amplifier will be linear. As shown in Figure 2-12, the linearizer samples the quadrature input signal $\mathbf{V}_m$ and the feedback quadrature demodulated feedback signal $\mathbf{V}_f$. According to these sampled signals, the linearizer generates a complex predistorted quadrature signal $\mathbf{V}_p$, which is used to linearize the output of the power amplifier.

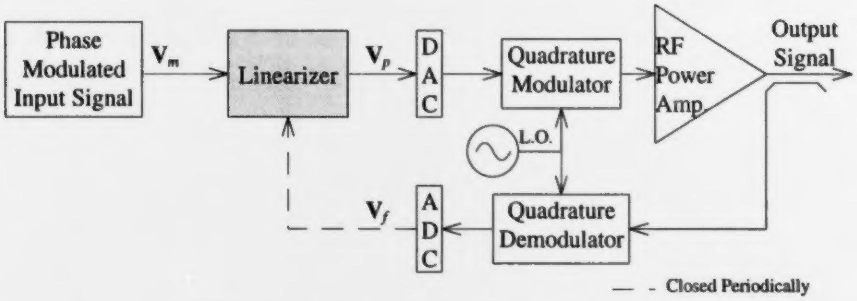
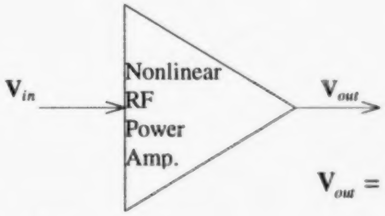

Phase Modulated Input Signal
V_m
Linearizer
V_p
DAC
Quadrature Modulator
RF Power Amp
Output Signal
L.O.
V_f
ADC
Quadrature Demodulator
Closed Periodically

Figure 2-12. Block diagram of adaptive predistortion technique.

Based on this approach, two processing techniques, namely, direct mapping predistortion [16] and complex gain predistortion [17, 18], have been proposed and studied. These techniques use a microprocessor to process the sampled signals and to generate the predistorted signal, based on a specific algorithm. The following subsections describe the details of these techniques.

2.3.4.1 Direct Mapping Predistortion

In this linearization technique, the distorted output $\mathbf{V}_{out}$ from a nonlinear power amplifier is obtained by adding the input signal of the amplifer $\mathbf{V}_{in}$ and a corresponding value of a distortion complex function $\mathbf{F}_d(\mathbf{V}_{in})$.



$$\mathbf{V}_{out} = K \cdot \left[\mathbf{F}_d(\mathbf{V}_{in}) + \mathbf{V}_{in}\right]$$

Figure 2-13. Model of a nonlinear power amplifier in direct mapping predistortion.

Figure 2-13 shows the model of the power amplifier, where K is the gain constant of the amplifier, $\mathbf{V}_{in}$ and $\mathbf{V}_{out}$ are the baseband equivalent complex input and output signal of the amplifier, respectively. Based on this model, the power amplifier can be linearized by using the following scheme:

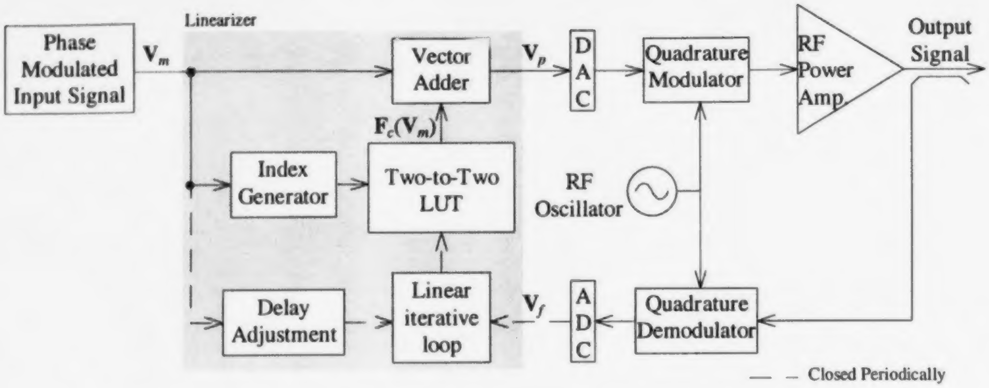

Linearizer
Phase Modulated Input Signal
$\mathbf{V}_m$
Vector Adder
$\mathbf{V}_p$
D A C
Quadrature Modulator
RF Power Amp
Output Signal
$\mathbf{F}_c(\mathbf{V}_m)$
Index Generator
Two-to-Two LUT
RF Oscillator
Delay Adjustment
Linear iterative loop
$\mathbf{V}_f$
A D C
Quadrature Demodulator
Closed Periodically

Figure 2-14. Block diagram of direct mapping predistortion.

The complex predistorted signal $\mathbf{V}_p$, generated by adding the quadrature input signal $\mathbf{V}_m$ to a corresponding complex correction function $\mathbf{F}_c(\mathbf{V}_m)$, is used to linearize the RF power amplifier. The values of $\mathbf{F}_c(\mathbf{V}_m)$ are stored in a LUT. With an ideal modulator, demodulator and D/A and A/D converter, the mathematical relations of the $\mathbf{F}_c$, $\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_d$ signals can be expressed as:

$$\begin{aligned} \mathbf{V}_p &= \mathbf{V}_m + \mathbf{F}_c(\mathbf{V}_m) \\ \mathbf{V}_f &= K \cdot \left[\mathbf{F}_d(\mathbf{V}_p) + \mathbf{V}_p\right] \end{aligned} \tag{2-1}$$

where $\mathbf{V}_f$ is the quadrature demodulated feedback signal, obtained from the quadrature demodulator. Substituting $-\mathbf{F}_d\left(\mathbf{V}_p\right)$ for $\mathbf{F}_c(\mathbf{V}_m)$, Eq. (2-1) can be re-written as follows:

$$\mathbf{V}_f = K \cdot \mathbf{V}_m \tag{2-2}$$

As a result, the output of the power amplifier can be linearized. However, using this technique, $\mathbf{F}_d(\mathbf{V}_p)$ cannot be found directly. The values of $\mathbf{F}_c(\mathbf{V}_m)$, in this case, are estimated by a linear iterative loop with respect to the difference between $\mathbf{V}_m$ and $\mathbf{V}_f$, i.e., the error signal $\mathbf{V}_{err}$ between $\mathbf{V}_m$ and $\mathbf{V}_f$. Eq. (2-3) shows the iterative equations for finding $\mathbf{F}_c(\mathbf{V}_m)$. The iteration is stopped when $\mathbf{V}_{err}$ is within a pre-determined tolerance.

$$\begin{aligned} \mathbf{V}_{err_{n,i}} &= \mathbf{V}_{m_{n,i}} - \mathbf{V}_{f_{n,i}} \\ \mathbf{F}_c\left(\mathbf{V}_{m_{n,i}}\right) &= \mathbf{F}_c\left(\mathbf{V}_{m_{n-1,i}}\right) + \zeta \cdot \mathbf{V}_{err_{n,i}} \end{aligned} \tag{2-3}$$

where n is the discrete time index, i is the sample index, ζ is the damping ratio of the linear iterative loop and $|\zeta| \leq 1$. Since $\mathbf{F}_c(\mathbf{V}_m)$ and $\mathbf{V}_m$ are complex signals and have a one-to-one relationship, a two-to-two LUT, indexed by $\mathbf{V}_m = I_m + j \cdot Q_m$ is required to map the entire range of $\mathbf{F}_c(\mathbf{V}_m) = I_c + j \cdot Q_c$. Moreover, this indicates that the L/S ratio, which is defined as the ratio of the number of points in the LUT to the number of sampling points, is equal to one, i.e., each set of the sampled data ($\mathbf{V}_{m_i}$ and $\mathbf{V}_{f_i}$) is required to pass through the iterative loop for evaluating a new $\mathbf{F}_c$ value. Therefore, this technique requires a large memory unit to store the LUT, and a very long update time to refresh the LUT. An experimental result obtained by Nagata [16] showed that this technique achieved approximately -60dB out-of-band power emission from a transmitter with a 20 Mbits memory unit and a 10 second update period.

2.3.4.2 Complex Gain Predistortion

This linearization technique [17, 18], shown in Figure 2-15, is similar to direct mapping predistortion. However, it effectively reduces the time for updating the LUT and the size of the LUT (i.e., from a two-to-two LUT to a one-to-two LUT) by applying the distortion mechanisms of a nonlinear RF power amplifier, described in Section 2.2.

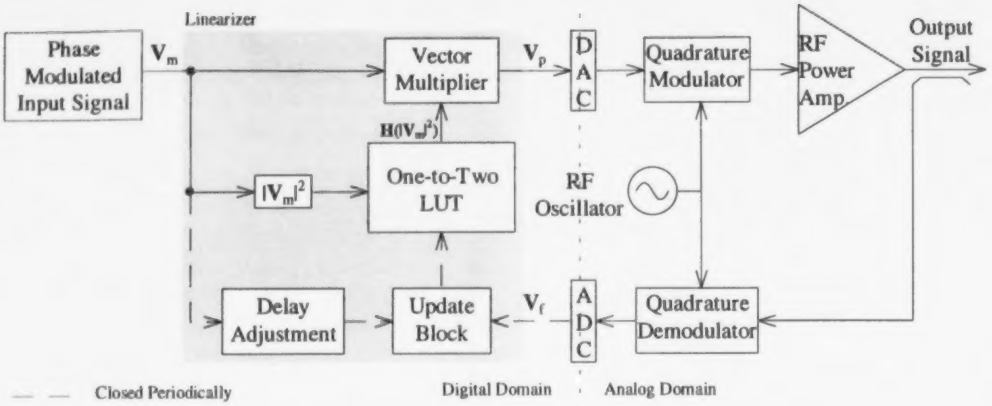

Linearizer
Phase Modulated Input Signal
V_m
Vector Multiplier
V_p
D A C
Quadrature Modulator
RF Power Amp
Output Signal
$H(|V_m|^2)$
$|V_m|^2$
One-to-Two LUT
RF Oscillator
Delay Adjustment
Update Block
V_f
A D C
Quadrature Demodulator
Closed Periodically
Digital Domain
Analog Domain

Figure 2-15. Block diagram of complex gain predistortion.

As shown in the block diagram of Figure 2-15, this linearization technique operates at baseband frequencies. The quadrature modulated input signal $\mathbf{V}_m = I_m + j \cdot Q_m$ and the quadrature demodulated feedback signal $\mathbf{V}_f$ are sampled synchronously, and these values are used to generate a predistortion vector function $\mathbf{H}(|\mathbf{V}_m|^2)$, which is stored in polar or rectangular form in a LUT. The input signal $\mathbf{V}_m$ is predistorted according to $\mathbf{H}(|\mathbf{V}_m|^2)$, so that the predistorted signal $\mathbf{V}_p$ produces the linearized output from the RF amplifier.

In this predistortion technique, the distortion of a nonlinear RF power amplifier is assumed to be characterized by memoryless distortion mechanisms, giving rise to amplitude and phase distortion that are dependent only on the amplitude of the input signal; i.e., AM-AM distortion and AM-PM distortion. This can be characterized by a complex gain function $\mathbf{G}(|\mathbf{V}_{in}|^2)$, so that the amplifier can be described mathematically by:

$$\mathbf{V}_{out} = \mathbf{V}_{in} \cdot \mathbf{G}\left(\left|\mathbf{V}_{in}\right|^2\right) \tag{2-4}$$

where $\mathbf{V}_{in}$ and $\mathbf{V}_{out}$ are the baseband equivalent complex input and output signal of the amplifier, respectively. Assuming an ideal modulator, demodulator, D/A and A/D converter, then from Eq. (2-4), the relationship between $\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$ can be expressed as:

$$\begin{aligned} \mathbf{V}_p &= \mathbf{V}_m \cdot \mathbf{H}\left(\left|\mathbf{V}_m\right|^2\right) \\ \mathbf{V}_f &= \mathbf{V}_p \cdot \mathbf{G}\left(\left|\mathbf{V}_p\right|^2\right) \end{aligned} \tag{2-5}$$

Therefore, in order for the overall amplifier characteristic to be linear, we require that

$$\mathbf{V}_f = K \cdot \mathbf{V}_m$$

so that

$$\mathbf{H}\left(\left|\mathbf{V}_m\right|^2\right) = \frac{K}{\mathbf{G}\left(\left|\mathbf{V}_p\right|^2\right)} \tag{2-6}$$

where K is a gain constant of the feedback loop. The linearization circuitry must therefore determine the predistortion function $\mathbf{H}(|\mathbf{V}_m|^2)$, and use this to generate the predistorted signal $\mathbf{V}_p$. In order to determine $\mathbf{H}(|\mathbf{V}_m|^2)$, three implementation algorithms, namely the secant method [17, 18], the Rotate and Scale (RASCAL) method [19] and the cubic spline interpolation method [4, 13] have been proposed. For each of these methods, the linear performance of the amplifier, expressed in terms of out-of-band power and intermodulation products, satisfies the North American TDMA digital cellular standard. The next subsections will describe these methods separately.

2.3.4.2.1 Secant Method

The secant method is a numerical root solving method, which is used to solve x in the function $f(x)=0$ if $f(x)$ is convergent. The iterative equation of this method is written as:

$$x_{i+1}=\frac{x_{i-1}\cdot f(x_i)-x_i\cdot f(x_{i-1})}{f(x_i)-f(x_{i-1})} \tag{2-7}$$

where i is the iterative index. Applying this method to the complex gain predistortion, the predistortion function $\mathbf{H}(|\mathbf{V}_m|^2)$ can be found by solving the complex error function $\mathbf{V}_{err}(\mathbf{H})$, i.e., the $f(x)$ in Eq. (2-7), which is defined as:

$$\mathbf{V}_{err}(\mathbf{H})=\mathbf{V}_f(\mathbf{H})-K\cdot\mathbf{V}_m=0 \tag{2-8}$$

where $\mathbf{H}$ is equivalent to $\mathbf{H}(|\mathbf{V}_m|^2)$. An experimental result obtained by A. S. Wright and W. G. Durtler [19] has shown that convergence of this method is sensitive to the initial conditions. Incorrect initial conditions prevent convergence and degrade the linearization performance. In addition, several iterative cycles for calculating a $\mathbf{H}(|\mathbf{V}_m|^2)$ value are required, and the L/S ratio is equal to one; this method suffers from an intensive computation requirement for finding the $\mathbf{H}(|\mathbf{V}_m|^2)$ function.

2.3.4.2.2 Rotate and Scale (RASCAL) Method

The RASCAL method [19] is similar to the linear iterative loop, which has been used in direct mapping predistortion, but with the difference that this method is used to estimate the predistortion transfer function $\mathbf{H}(|\mathbf{V}_m|^2)$ rather than $\mathbf{V}_c(\mathbf{V}_m)$. This method calculates a scale error value e_{scale}, and a phase error value e_{rotate} for each sampled $\mathbf{V}_m$ and $\mathbf{V}_f$ with respect to $|\mathbf{V}_m|^2$. Eq. (2-9) shows the definition of the e_{scale} and e_{rotate}.

$$\begin{aligned} e_{scale}\left(|\mathbf{V}_m|^2\right) &= |\mathbf{V}_f| - |\mathbf{V}_m| \\ e_{rotate}\left(|\mathbf{V}_m|^2\right) &= \arg(\mathbf{V}_f) - \arg(\mathbf{V}_m) \end{aligned} \qquad (2\text{-}9)$$

Applying e_{scale} and e_{rotate} to the iterative linear algorithm [19], i.e., the linear iterative loop, the linearizer can estimate the $\mathbf{H}(|\mathbf{V}_m|^2)$ function, and also generates a new LUT. Eq. (2-10) and Figure 2-16 show the definition of the iterative linear algorithm, and a block diagram for using this method in complex gain predistortion.

$$\begin{aligned} S_{n+1} &= S_n - \zeta \cdot e_{scale}\left(|\mathbf{V}_m|^2\right) \\ R_{n+1} &= R_n - \zeta \cdot e_{rotate}\left(|\mathbf{V}_m|^2\right) \\ \mathbf{H}\left(|\mathbf{V}_m|^2\right) &= S \cdot e^{j \cdot R} \end{aligned} \qquad (2\text{-}10)$$

where n is the discrete time index, and ζ is the damping ratio of the iterative linear algorithm and $|\zeta|<1$.

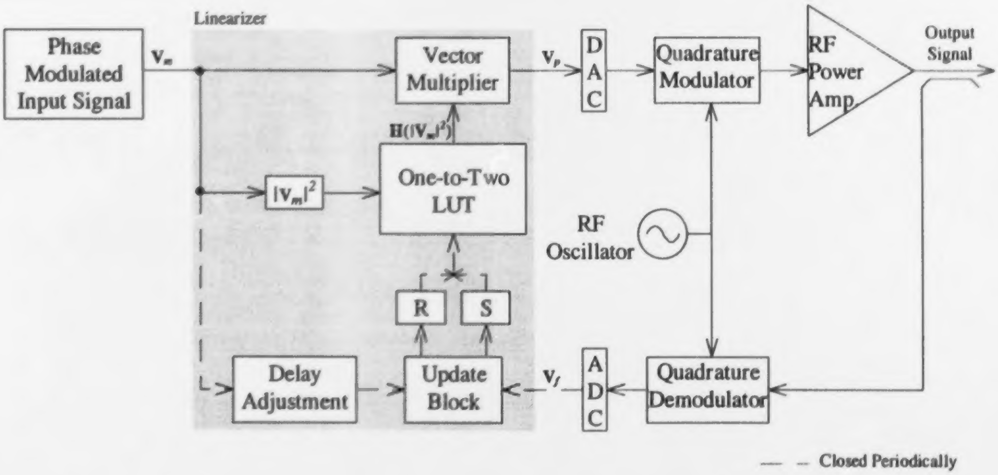

Linearizer
Phase Modulated Input Signal
$\mathbf{V}_m$
Vector Multiplier
$\mathbf{V}_p$
D A C
Quadrature Modulator
RF Power Amp
Output Signal
$\mathbf{H}(|\mathbf{V}_m|^2)$
$|\mathbf{V}_m|^2$
One-to-Two LUT
RF Oscillator
R
S
Delay Adjustment
Update Block
$\mathbf{V}_f$
A D C
Quadrature Demodulator
Closed Periodically

Figure 2-16. Block diagram of complex gain predistortion using RASCAL method.

The iterations are continued until the values of e_{scale} and e_{rotate} are within a specified tolerance. The linear algorithm is independent of the initial condition and is stable if $|\zeta|<1$. The divergence that occurs in the secant method can be avoided. An experimental result, also obtained by A. S. Wright and W. G. Durtler, acheived acceptable (IS-54 specifications) linearization of a class B power amplifier with a 64 point LUT and an input data bit rate of 2 kbps. However, this method requires intensive computation to perform the conversion between rectangular coordinates and polar coordinates for each set of the sampled signals ($\mathbf{V}_m$ and $\mathbf{V}_f$).

2.3.4.2.3 Cubic Spline Interpolation Method

Instead of calculating the predistortion function $\mathbf{H}(|\mathbf{V}_m|^2)$ and mapping it into the LUT, this method applies cubic spline interpolation and a complex mathematical function $\mathbf{C}(|\mathbf{V}_p|^2)$ to generate a LUT [4, 13]. Eq. (2-11) gives the definition of $\mathbf{C}(|\mathbf{V}_p|^2)$:

$$\begin{aligned} &\mathbf{C}\left(|\mathbf{V}_p|^2\right) = |\mathbf{V}_f| \cdot e^{j\cdot\Delta\Phi} \\ \text{where} \quad &\Delta\Phi = \arg(\mathbf{V}_f) - \arg(\mathbf{V}_p) \end{aligned} \tag{2-11}$$

In Figure 2-17, one implementation of this method to achieve complex gain predistortion is shown.

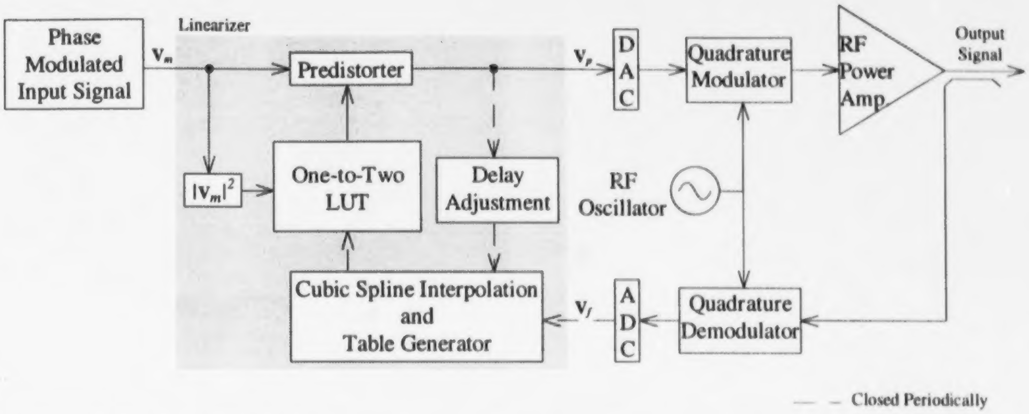

Phase Modulated Input Signal
Linearizer
$\mathbf{V}_m$
Predistorter
$\mathbf{V}_p$
D A C
Quadrature Modulator
RF Power Amp
Output Signal
$|\mathbf{V}_m|^2$
One-to-Two LUT
Delay Adjustment
RF Oscillator
Cubic Spline Interpolation and Table Generator
$\mathbf{V}_f$
A D C
Quadrature Demodulator
Closed Periodically

Figure 2-17. Block diagram of complex gain predistortion using cubic spline interpolation method.

As shown in Eq. (2-11), $\mathbf{C}(|\mathbf{V}_p|^2)$ is a function of the magnitude of $\mathbf{V}_f$ and the phase shift between $\mathbf{V}_p$ and $\mathbf{V}_f$, and is indexed by the magnitude squared of $\mathbf{V}_p$. When the linearizer has collected enough data samples from $\mathbf{V}_p$ and $\mathbf{V}_f$, cubic spline interpolation uses all of this data to construct $\mathbf{C}(|\mathbf{V}_p|^2)$. The table generator then applies the relationship of Eq. (2-12) to generate a new LUT, in which the magnitude of the new predistorted signal $|\mathbf{V}_{p_{LUT}}|$, and the corresponding phase shift $\Delta\phi_{LUT}$ are stored.

$$C^{-1}\left(|V_f|^2\right) \equiv inverse\left(\left|\mathbf{C}\left(|\mathbf{V}_p|^2\right)\right|\right)$$

$$\left|\mathbf{V}_{p_{LUT}}\right| = \left|C^{-1}\left(|\mathbf{V}_m|^2\right)\right| \qquad (2\text{-}12)$$

$$\Delta\Phi_{LUT} = \arg\left(\mathbf{C}\left(\left|\mathbf{V}_{p_{LUT}}\right|^2\right)\right)$$

Since the values contained in the LUT are not $\mathbf{H}(|\mathbf{V}_m|^2)$, the linearizer cannot directly multiply the $\mathbf{V}_m$ signal by the corresponding value in the LUT. In this case, the input $\mathbf{V}_m$ signal is fed into a predistorter. This predistorter applies Eq. (2-13) to generate the required predistorted signal $\mathbf{V}_p$.

$$\mathbf{V}_p = \left|\mathbf{V}_{p_{LUT}}\right| \cdot e^{j(\arg(\mathbf{V}m)+\Delta\Phi_{LUT})} \tag{2-13}$$

where $\mathbf{V}_{p_{LUT}}$ and $\Delta\Phi_{LUT}$ are the corresponding LUT values, indexed by the $|V_m|^2$.

The cubic spline interpolation method achieves a high L/S ratio and effectively eliminates the instability due to improper initial conditions. Simulation results obtained by A. Lohtia [4, 13] have shown that this method achieved approximately -80dB out-of-band power emission from the output of a class B power amplifier with 30 spline points and 10,000 points in the LUT (i.e., L/S ratio = 333.33). However, because of the complexity of the cubic spline interpolation and the polar coordinate manipulations, this method requires intensive computation. In addition, this method is very sensitive to signal synchronization, unequal spacing of the spline points and the gain constant of the feedback loop (K).

2.4 Summary of Nonlinear RF Amplifier Linearization Techniques

All of the linearization techniques mentioned in Section 2.3 have appeared in the literature [3, 5, 7, 13, 15, 16]; for all of thses methods, the performance in terms of the out-of-band power and intermodulation products satisfies the IS-54 specification (see Section 1.1). Table 2-1 summarizes the characteristics of each linearization technique. According to Table 2-1, only two of these techniques (cartesian coordinate negative feedback and adaptive predistortion) can adaptively compensate for changes in the operating environment such as RF channel switching and power level control, and changes in ambient and circuit conditions. Cartesian coordinate negative feedback is a combined RF and baseband closed loop feedback technique. Therefore, the loop time delay limits the usable bandwidth. On the other hand, adaptive predistortion is relatively insensitive to amplifier variations and RF channel switching. This technique may therefore be the most suitable one for implementing a linearized transmitter for cellular mobile radio application.

Two digitally adaptive predistortion techniques have been developed: direct mapping predistortion and complex gain predistortion. Complex gain predistortion requires a smaller memory, and achieves a faster update cycle time than direct mapping predistortion. Three algorithms based on the complex gain linearization technique, namely the secant method, the Rotate and Scale (RASCAL) method and the cubic spline interpolation method, have been proposed and studied. However, these methods suffer from one or more problems, namely sensitivity to initial conditions, intensive computation requirements that limit the input signal bit rate, and low L/S ratio. In the next chapter, an enhanced linearization method based on the complex gain predistortion technique is proposed; the implementation of the enhanced linearizer will be described in Chapter 4.

Table 2-1. SUMMARY OF THE PERFORMANCE CHARACTERISTICS OF SEVERAL LINEARIZATION TECHNIQUES.

Technique	Implementation	Advantages	Disadvantages
Feedforward correction	Analog and open loop system.	• Unconditionally stable • Using highly efficient nonlinear amplifier	• Matching of power requirements of main and auxiliary amplifier, and circuit complexity • No compensation of characteristics for temperature change and aging • Primarily reduces only first order nonlinearities
LINC/ CALLUM	Analog and open loop system.	• Unconditionally stable • Simple implementation • Using highly efficient nonlinear amplifier	• Matching of amplifier characteristics • Generation of 2 constant amplitude signals • Loop gain control, bandwidth restrictions • No compensation of characteristics for temperature change and aging
Cartesian coordinate negative feedback	Analog and closed loop system.	• Adaptive compensation • Circuit simplicity	• Loop gain control • Bandwidth restrictions
Adaptive predistortion (Direct Mapping Predistortion)	Digital and semi closed loop system.	• Adaptive compensation • Stability • Simple algorithm	• Large memory requirement • Long update time • Low L/S ratio
Adaptive predistortion (complex gain predistortion using secant method)	Digital and semi closed loop system.	• Adaptive compensation • Small memory unit • Faster update time	• Instability • Intensive computation • Low L/S ratio
Adaptive predistortion (complex gain predistortion using RASCAL method)	Digital and semi closed loop system.	• Adaptive compensation • Small memory unit • Faster update time • Stability	• Intensive computation • Low L/S ratio
Adaptive predistortion (complex gain predistortion using cubic spline interpolation method)	Digital and semi closed loop system.	• Adaptive compensation • High L/S ratio • Stability	• Large memory requirement • Intensive computation • Feedback signal synchronization

2.5 Impairments in the Quadrature Modulator and Demodulator

The quadrature modulator and demodulator provide the interface between the baseband I and Q signals and the transmitted RF signal. These components are not ideal, but suffer from three major impairments, namely d.c. offset, gain imbalance and phase imbalance [3, 4, 20, 21, 22, 23]. d.c. offset is caused by carrier leakage inside the mixer, and is equivalent to a linear shift of the signal constellation from its original position. Gain imbalance is defined as the gain ratio between I and Q signals. This impairment causes deformation of the signal constellation pattern, and changes the circular shape to an ellipse. Phase imbalance is caused by two sources of error (the phase difference in the 90° phase splitter and the difference of the signal paths), and results in the whole signal constellation being slanted to a certain degree. Figure 2-18 illustrates the effects of these impairments on a sinusoidal input applied to the I and Q signals (i.e., the I signal is a cosine wave and the Q signal is a sine wave).

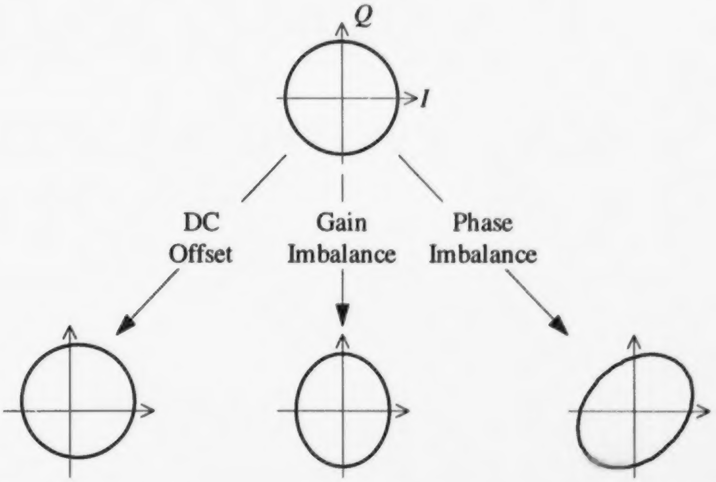


Figure 2-18. Effects of the quadrature modulator and demodulator impairments (DC offset, gain imbalance and phase imbalance) on sinusoidal I and Q input signals.

The impairments in the quadrature modulator and demodulator generate spurious frequency components in the undesired sideband. These spurious frequency components interact with the distortions of the nonlinear RF power amplifier (i.e., AM-AM distortion and AM-PM distortion) and generate intermodulation distortion products. As a result, the performance of the whole digital transmitter will be degraded.

Several methods [24, 25, 26, 27] has been proposed to correct the impairments in the quadrature modulator and demodulator. A digital method for correcting the quadrature modulator impairments is described by D. Hillborn, *et al.* [28]. This method neglects the effect due to the imperfection of the demodulator and uses a vector compensator to correct the impairments in the modulator. The vector compensator multiplies a constant correction vector by the predistorted signal $\mathbf{V}_p$ in order to compensate the imperfection in the quadrature modulator. The value of the correction vector is obtained by pre-calibrating all of the impairment parameters of the modulator beforehand. Because this method only requires vector multiplication to correct the modulator's impairment, it is achievable by using digital signal processing in real time. However, this method cannot compensate the modulator adaptively due to its open loop characteristic.

Another digital method, proposed by A. Lohtia, *et al.* [4, 25], applies two subsidiary loops, namely the quadrature modulator compensator and the demodulator impairment compensator, around the modulator and demodulator, separately. Figure 2-19 shows a block diagram of this method.

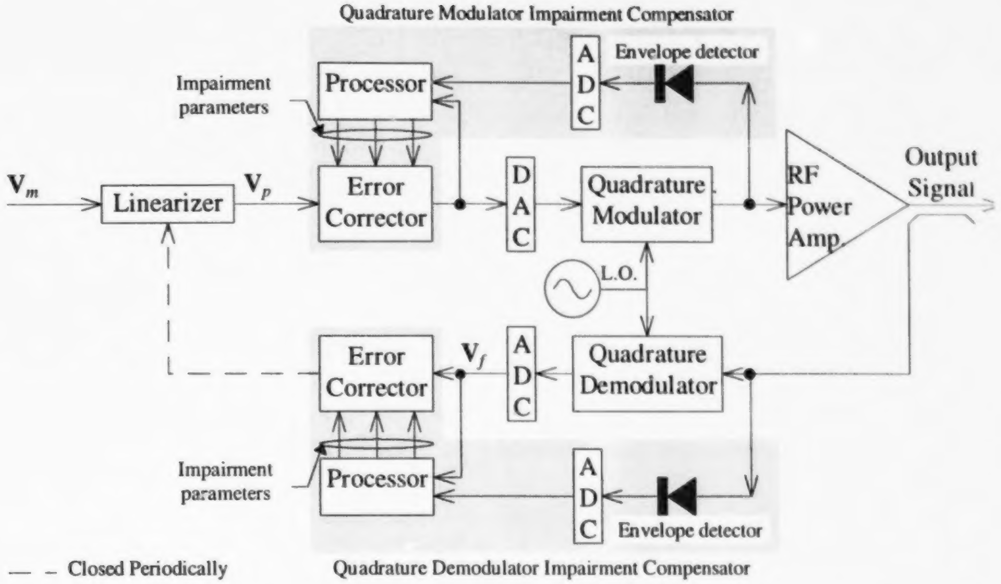

Quadrature Modulator Impairment Compensator
Impairment parameters
Processor
A D C
Envelope detector
$\mathbf{V}_m$
Linearizer
$\mathbf{V}_p$
Error Corrector
D A C
Quadrature Modulator
RF Power Amp.
Output Signal
L.O.
Error Corrector
$\mathbf{V}_f$
A D C
Quadrature Demodulator
Impairment parameters
Processor
A D C
Envelope detector
Closed Periodically
Quadrature Demodulator Impairment Compensator

Figure 2-19. Block diagram of quadrature modulator and demodulator impairment compensator.

As shown in Figure 2-19, the feedback RF signal, i.e., the output of the quadrature modulator and the output of the power amplifier, are detected by the envelope detectors individually. The detected signal ideally is equal to the instantaneous magnitude of the feedback RF signal. Two A/D converters are used to sample the detected signals, which are then sent as digitized signals to the processors. The processors synchronously sample the corresponding predistorted signal $\mathbf{V}_p$ and the quadrature demodulated signal $\mathbf{V}_f$. When five sets of uncorrelated data have been sampled, each processor uses the sampled data to form five nonlinear equations, and then applies the Newton-Raphson method to solve for all of the impairment parameters. These impairment parameters are sent to the error correctors to correct the modulator and demodulator imperfections. This method can adaptively compensate the impairments in the modulator and demodulator. However, the drawbacks of this method are that it requires large computational power to estimate the impairment parameters, and requires high precision components to detect the feedback RF signals.

According to the specifications of a quadrature modulator and demodulator (see Table 2-2) provided by Mini-Circuits [29], the impairments of this quadrature modulator and demodulator are small, especially for the quadrature demodulator. For cellular mobile radio applications, it is difficult and expensive to design a detection circuit which can sense these small errors. In this case, the open loop method, used by D. Hillborn, *et al.*, is a practical approach to compensate digitally for the impairments in the quadrature modulator and demodulator.

Table 2-2. SPECIFICATION OF A QUADRATURE MODULATOR AND DEMODULATOR.

	Frequency (MHz)	Amplitude imbalance (dB) typ.	Phase imbalance (Degrees) typ.
Quadrature modulator (QMC-1050)	800-1050	0.5	3
Quadrature demodulator (MIQA-895D)	868-895	0.15	1.5

3. ENHANCED LINEARIZATION METHOD

In this chapter, a linearization method is proposed which is based on the complex gain predistortion technique. The aim is to achieve a practicable linearizer with a short look-up-table (LUT) update time, a relatively high L/S ratio requiring a small amount of memory, and moderate computational requirement. Figure 3-1 shows a block diagram of a digital transmitter with the enhanced linearizer.

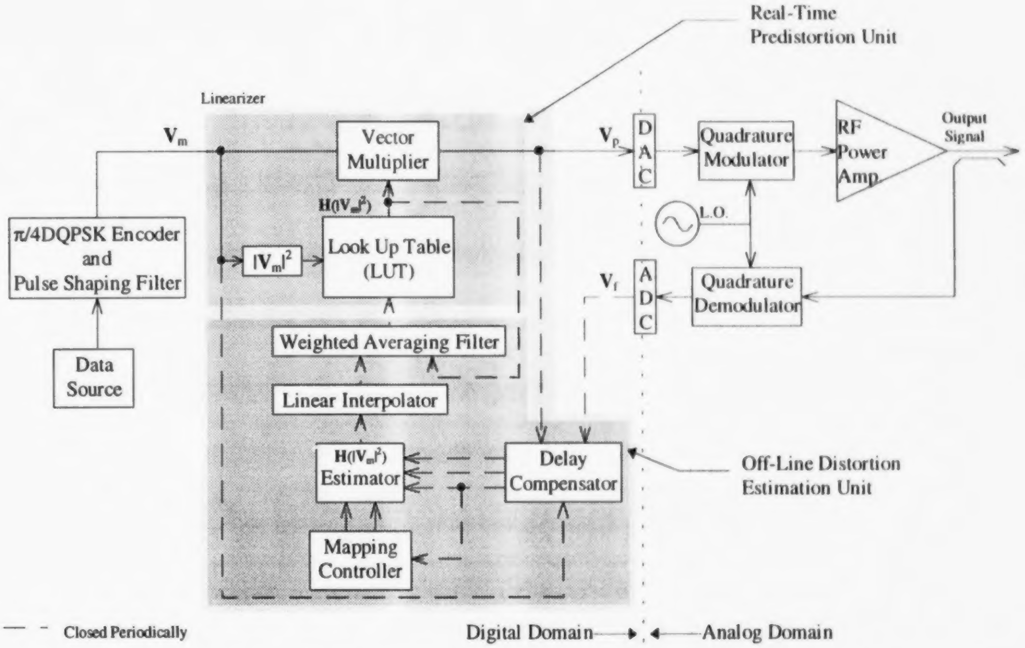

Real-Time
Predistortion Unit
Linearizer
V_m
Vector Multiplier
$H(|V_m|^2)$
π/4DQPSK Encoder and Pulse Shaping Filter
$|V_m|^2$
Look Up Table (LUT)
Weighted Averaging Filter
Data Source
Linear Interpolator
$H(|V_m|^2)$ Estimator
Delay Compensator
Mapping Controller
V_p
DAC
Quadrature Modulator
RF Power Amp
Output Signal
L.O.
V_f
ADC
Quadrature Demodulator
Off-Line Distortion
Estimation Unit
Closed Periodically
Digital Domain
Analog Domain

Figure 3-1. Block diagram of the enhanced linearization method.

The enhanced linearizer consists of two basic units: a real-time predistortion unit, and an off-line distortion estimation unit. In the real-time predistortion unit, the input quadrature signal $\mathbf{V}_m$, obtained from the output of the π/4 DQPSK encoder and pulse shaping filter, is multiplied by a corresponding predistortion vector. The predistortion vector is stored in a LUT which is indexed by the magnitude squared of $\mathbf{V}_m$. The resulting complex signal from the vector multiplier is the predistorted signal $\mathbf{V}_p$. After D/A conversion, the RF signal is generated using a quadrature modulator. This RF signal is amplified by the power amplifier to produce the final output. This final output has an overall linear RF response with respect to the original input signal $\mathbf{V}_m$. The core of the enhanced linearizer is the off-line distortion estimation unit which consists of five parts: a delay compensator, the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, a mapping controller, a linear interpolator, and a weighted averaging filter. The delay compensator samples three digital signals ($\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$), and provides synchronous samples to the mapping controller and the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator. The mapping controller selects N sampled sets of $\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$ from the output of the delay compensator. These N sampled sets cover the entire range of the input signal $\mathbf{V}_m$. The $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, controlled by the mapping controller, latches the selected sets of data, and calculates N values of $\mathbf{H}(|\mathbf{V}_m|^2)$. These values of $\mathbf{H}(|\mathbf{V}_m|^2)$ characterize the amplifier over its entire operating range, and are passed to the linear interpolator and weighted averaging filter to construct M averaged predistortion values, which are then stored in the predistortion LUT. The following sections will describe the functions and the characteristics of the delay compensator, the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, the mapping controller, the linear interpolator and the weighted averaging filter.

3.1 Delay Compensator

The delay compensator is used to synchronize the three received signals ($\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$). Since $\mathbf{V}_m$ and $\mathbf{V}_p$ are already synchronized to the sampling clock, they need not be corrected. However, $\mathbf{V}_f$ suffers a time delay, originating from the reconstruction filter in the DAC, the anti-aliasing filter in the ADC, as well as the quadrature modulator and demodulator. The delay of the RF amplifier is included in its complex gain function. With the assumption that the delay of the $\mathbf{V}_f$ signal does not vary too much with signal frequency (i.e., an almost constant time delay), and is composed of two types of time delay: discrete sampling time delay (τ_d) and fractional time delay (τ_f), the $\mathbf{V}_f$ signal can be adjusted by using a delay adjusting buffer and linear interpolation, i.e., the $\mathbf{V}_f$ values used to compute $\mathbf{H}(|\mathbf{V}_m|^2)$ are obtained from the actual $\mathbf{V}_f$ values coming from the ADC using a storage buffer and interpolation. The delay adjusting buffer is used to correct the τ_d in the $\mathbf{V}_f$ signal, and provides an adjusted $\mathbf{V}_f$ signal which has a time delay less than one sampling interval. Linear interpolation is then used to correct the τ_f in the adjusted $\mathbf{V}_f$ signal. In this way, a corrected $\mathbf{V}_f$ signal is obtained from the delay compensator that is synchronous with the $\mathbf{V}_m$ and $\mathbf{V}_p$ signals.

Since the time delay in $\mathbf{V}_f$ is assumed to be constant, the values of τ_d and τ_f can be pre-determined before the linearization process starts. The value of τ_d is easily calculated from the group delay characteristic of the filters in the DAC and ADC, quadrature modulator, and demodulator. If the averaged group delay of each filter in the DAC and ADC is roughly equal to τ_{filter}, and the delay of the quadrature modulator and demodulator is insignificant with respect to that of the filters, then the value of τ_d can be estimated using the following equation:

$$\tau_d = RND(\frac{2 \cdot \tau_{filter}}{T_s}) \cdot T_s \tag{3-1}$$

where *RND* is the round-up function which returns a closest integer value from its argument, and T_s is the sampling cycle. Because the value of τ_d is a multiple of T_s, the delay adjusting buffer should have a length of τ_d/T_s in order to correct the τ_d in $\mathbf{V}_f$. The value of τ_f can be obtained by

$$\tau_f = 2 \cdot \tau_{filter} - \tau_d \tag{3-2}$$

However, since τ_{filter} is a roughly estimated value, and the phase characteristics of the filters in the I and Q channels are not identical, the value of τ_f cannot be determined by Eq. (3-2); in addition, two different fractional time delay values (τ_I and τ_Q) are required for representing the τ_f in the quadrature components (I_f and Q_f) of $\mathbf{V}_f$ separately. The minimum mean square error (MMSE) timing recovery algorithm [30] is used to calibrate τ_f in the I and Q channels, in order to obtain the value of the τ_I and τ_Q. Eq. (3-3) gives the mathematical expressions for using the MMSE algorithm to obtain τ_f for both the I and Q channels:

$$\begin{aligned} \tau_{I_{i+1}} &= \tau_{I_i} - k \cdot \left(I_{f_i}(\tau) - I_{P_i}\right) \cdot \left(I_{f_{i-2}}(\tau) - 2 \cdot I_{f_{i-1}}(\tau) + 2 \cdot I_{f_{i+1}}(\tau) - I_{f_{i+2}}(\tau)\right) \\ \tau_{Q_{i+1}} &= \tau_{Q_i} - k \cdot \left(Q_{f_i}(\tau) - Q_{P_i}\right) \cdot \left(Q_{f_{i-2}}(\tau) - 2 \cdot Q_{f_{i-1}}(\tau) + 2 \cdot Q_{f_{i+1}}(\tau) - Q_{f_{i+2}}(\tau)\right) \end{aligned} \tag{3-3}$$

where i is the sampling index, τ_I and τ_Q are corresponding to the τ_f in the I and Q channels, and k is the damping coefficient. The detailed derivation of Eq. (3-3) is shown in Appendix A. Since the values of τ_I and τ_Q do not include the distortion of the power amplifier, the calibration must take place without connecting the power amplifier. The value of τ_f in I_f and Q_f can now be corrected by separately applying the calibrated value of the τ_I and τ_Q to the linear interpolation. Eq. (3-4) shows the mathematical expressions of this correction process:

$$\begin{aligned} I_{f_0}' &= I_{f_0} + \frac{\tau_I}{T_s}\left(I_{f_1} - I_{f_0}\right) \\ Q_{f_0}' &= Q_{f_0} + \frac{\tau_Q}{T_s}\left(Q_{f_1} - Q_{f_0}\right) \end{aligned} \tag{3-4}$$

where I_f' and Q_f' are the corrected quadrature signals of $\mathbf{V}_f$, and the subscripts of 0 and 1 are represented the data without discrete time delay and the data delayed by one sampling cycle from the previous data, respectively.

3.2 H(|V$_m$|^2) Estimator

The $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator is used to estimate the predistorter's transfer function $\mathbf{H}(|\mathbf{V}_m|^2)$. Referring to Section 2.3.4.2, the complex gain predistortion technique can be formulated as:

$$\mathbf{V}_p = \mathbf{V}_m \cdot \mathbf{H}\left(\left|\mathbf{V}_m\right|^2\right)$$

and $$\mathbf{V}_f = \mathbf{V}_p \cdot \mathbf{G}\left(\left|\mathbf{V}_p\right|^2\right)$$

so $$\mathbf{V}_f = K \cdot \mathbf{V}_m$$

requiring $$\mathbf{H}\left(\left|\mathbf{V}_m\right|^2\right) = \frac{K}{\mathbf{G}\left(\left|\mathbf{V}_p\right|^2\right)} \tag{3-5}$$

where $\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$ are digital signals representing the quadrature input signal, the complex predistorted signal and the quadrature demodulated feedback signal respectively,

K is the gain constant of the power amplifier, and $\mathbf{G}(|\mathbf{V}_p|^2)$ is the normalized complex gain function of the power amplifier. From Eq. (3-5), it may easily be shown that

$$\mathbf{H}\left(\left|\mathbf{V}_m\right|^2\right) = K \cdot \frac{\mathbf{V}_p \cdot \mathbf{V}_f^{\,*}}{\left|\mathbf{V}_f\right|^2} \tag{3-6}$$

where * is the complex conjugate operation. As shown in (3-6), the estimator samples the three digital signals ($\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$) and evaluates a corresponding value for the $\mathbf{H}(|\mathbf{V}_m|^2)$ function. The $\mathbf{V}_p$ and $\mathbf{V}_f$ are independent variables of the $\mathbf{H}(|\mathbf{V}_m|^2)$ function, and the $\mathbf{V}_m$ is the index variable. For DSP implementation, the computational complexity is reduced if rectangular coordinates are used in (3-6), instead of polar coordinates. Therefore, let $\mathbf{V}_f = I_f + j \cdot Q_f$, and $\mathbf{V}_p = I_p + j \cdot Q_p$. Then (3-6) can be rewritten as:

$$\mathbf{H}\left(\left|\mathbf{V}_m\right|^2\right) = \alpha\left(\left|\mathbf{V}_m\right|^2\right) + j \cdot \beta\left(\left|\mathbf{V}_m\right|^2\right)$$

$$\alpha\left(\left|\mathbf{V}_m\right|^2\right) = K \cdot \frac{I_f \cdot I_p + Q_f \cdot Q_p}{\left|\mathbf{V}_f\right|^2}\ , \tag{3-7}$$

and

$$\beta\left(\left|\mathbf{V}_m\right|^2\right) = K \cdot \frac{I_f \cdot Q_p - Q_f \cdot I_p}{\left|\mathbf{V}_f\right|^2}\ .$$

Thus, as shown in (3-7), two simple algebraic equations, $\alpha(|\mathbf{V}_m|^2)$ and $\beta(|\mathbf{V}_m|^2)$, with $|\mathbf{V}_m|^2$ as the reference are used to calculate the $\mathbf{H}(|\mathbf{V}_m|^2)$ function. Because of the simplicity of these equations and the relatively small computational requirement, a shorter update time for the LUT can be achieved in the realization of the linearizer.

3.3 Mapping Controller

The delay compensator and the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator were assumed to operate continuously in order to generate a $\mathbf{H}(|\mathbf{V}_m|^2)$ function which covers the entire value range of $|\mathbf{V}_m|^2$. Since the variation of adjacent sets of sampled data is very small, it is not necessary to evaluate every input data set; this would result in a waste of computational resources. One approach is to divide the entire value range of $|\mathbf{V}_m|^2$ into N equal intervals. In each interval, only one set of sampled signals ($\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$) is used for calculating a value of $\mathbf{H}(|\mathbf{V}_m|^2)$. Because there are N sets of data for evaluation, the amount of computation needed for calculating the $\mathbf{H}(|\mathbf{V}_m|^2)$ function can be significantly reduced. Moreover, the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator can obtain all of the required samples for different power levels of $\mathbf{V}_m$ signal without checking every input.

The mapping controller is used to implement this approach. It generates two outputs corresponding to a given $|\mathbf{V}_m|^2$ input value: an enable signal and a sampling index. The enable signal is used to activate the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator to latch the appropriate set of $\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$ signals. The sampling index, calculated from the controller's mapping function $M(|\mathbf{V}_m|^2)$, is used to allocate a memory block for storing the corresponding latched values $\hat{\mathbf{V}}_m$, $\hat{\mathbf{V}}_p$ and $\hat{\mathbf{V}}_f$. Since the entire value range of $|\mathbf{V}_m|^2$ is divided into N equal intervals, $M(|\mathbf{V}_m|^2)$ can be defined as:

$$M\left(\left|\mathbf{V}_m\right|^2\right) \equiv \left\lfloor N \cdot \frac{\left|\mathbf{V}_m\right|^2}{\left|\mathbf{V}_m\right|^2_{MAX}} \right\rfloor \qquad N \geq 1$$

$$i = M\left(\left|\mathbf{V}_m\right|^2\right) \qquad i \in [0 \cdots N-1] \tag{3-8}$$

where $\lfloor\ \rfloor$ is a *floor* operator which returns the greatest integer less than or equal to its argument, $|\mathbf{V}_m|^2_{\max}$ is the maximum value of $|\mathbf{V}_m|^2$, and i is the sampling index. Since the distribution of the $|\mathbf{V}_m|^2$ values is not even, the controller may take very long time to collect a set of data which will include values over the whole input range. In order to solve this searching problem, the controller should divide the entire value range of $|\mathbf{V}_m|^2$ into N unequal spacing intervals in such a way that the occurrence probability of the signal in each interval is the same. With the assumption that the transmitted digital information is randomly distributed (i.e., the probability of occurrence of a 0 or a 1 are equal), and that $|\mathbf{V}_m|^2$ has a fixed probability distribution function (pdf), the $M(|\mathbf{V}_m|^2)$ and the sampling index (i) can be re-defined as in (3-9). Because the pdf is fixed and is also a function of $|\mathbf{V}_m|^2$, a constant relationship between $|\mathbf{V}_m|^2$ and i, as shown in Figure 3-2, can be maintained.

$$\begin{aligned} M\left(\left|\mathbf{V}_m\right|^2\right) &\equiv \left\lfloor N \cdot \int_0^{|\mathbf{V}_m|^2} pdf\left(A_m^{\,2}\right) d\left(A_m^{\,2}\right) \right\rfloor \\ &\equiv \left\lfloor N \cdot cdf\left(\left|\mathbf{V}_m\right|^2\right) \right\rfloor \qquad N \geq 1 \end{aligned} \tag{3-9}$$

$$i \equiv M\left(\left|\mathbf{V}_m\right|^2\right) \qquad i \in [0 \cdots N-1]$$

where $A_m^{\,2}$ is the magnitude squared of $\mathbf{V}_m$, cdf is the cumulative distribution function of $|\mathbf{V}_m|^2$ and N is the number of intervals in the entire range of $|\mathbf{V}_m|^2$.

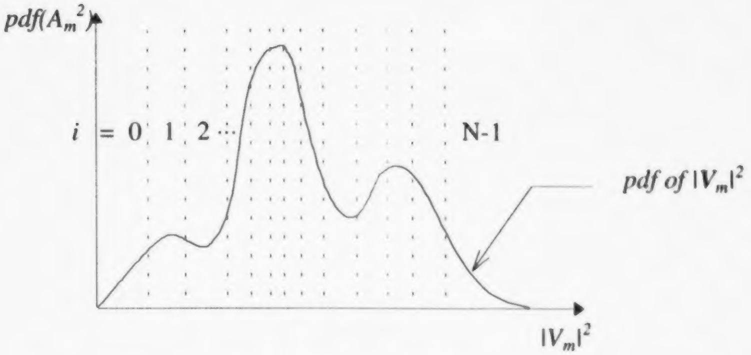


Figure 3-2. Relationship between $|\mathbf{V}_m|^2$ and sampling intervals, i.

As shown in (3-9) and Figure 3-2, the whole value range of $|\mathbf{V}_m|^2$ is divided into N equal probability intervals; i.e., the area of each interval under the pdf curve is the same, and each interval has its corresponding i value. In this case, the relationship between $|\mathbf{V}_m|^2$ and i can be mapped into a mapping table. Using this table, the mapping controller generates a value of i for a corresponding interval. Moreover, the enable signal is also asserted once for every i within one LUT update cycle time; i.e., the first occurred i in one interval is used to generate the enable signal. The $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, therefore, accumulates N sets of $\hat{\mathbf{V}}_m$, $\hat{\mathbf{V}}_p$ and $\hat{\mathbf{V}}_f$ values with unequal spacing, but with equal probability of occurrence of each $\left|\hat{\mathbf{V}}_m\right|^2$. Because there are only N sets of data for evaluation, the characteristic function of the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator (3-7) can be rewritten as (3-10):

$$\mathbf{H}_i\left(\left|\hat{\mathbf{V}}_{m_i}\right|^2\right)=\alpha_i\left(\left|\hat{\mathbf{V}}_{m_i}\right|^2\right)+j\cdot\beta_i\left(\left|\hat{\mathbf{V}}_{m_i}\right|^2\right)$$

where $$\alpha_i\left(\left|\hat{\mathbf{V}}_{m_i}\right|^2\right)=K\cdot\frac{\hat{I}_{f_i}\cdot\hat{I}_{p_i}+\hat{Q}_{f_i}\cdot\hat{Q}_{p_i}}{\left|\hat{\mathbf{V}}_{f_i}\right|^2} \tag{3-10}$$

and $$\beta_i\left(\left|\hat{\mathbf{V}}_{m_i}\right|^2\right)=K\cdot\frac{\hat{I}_{f_i}\cdot\hat{Q}_{p_i}-\hat{Q}_{f_i}\cdot\hat{I}_{p_i}}{\left|\hat{\mathbf{V}}_{f_i}\right|^2} \qquad i\in[0\cdots N-1]$$

where i is the sampling index and the symbol of ^ represents the value latched by the estimator. Using (3-10) for calculating the $\mathbf{H}(|\mathbf{V}_m|^2)$ function in the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, the amount of computation can be significantly reduced, and all of the required samples for different power levels of $\mathbf{V}_m$ signal can be obtained. In addition, the delay compensator does not need to adjust every set of sampled signals; it only corrects the one which is used for evaluating a value of the $\mathbf{H}(|\mathbf{V}_m|^2)$.

3.4 Linear Interpolator and Weighted Averaging Filter

The discrete and unequally separated values obtained from the estimator do not provide sufficient accuracy and resolution to represent the characteristic of the predistortion function $\mathbf{H}(|\mathbf{V}_m|^2)$, especially when the number of intervals N is small. Quantization noise and the different amplitude intervals between successive samples also degrade the accuracy of $\mathbf{H}(|\mathbf{V}_m|^2)$. Although increasing the value of N can improve the resolution or the accuracy of the calculated $\mathbf{H}(|\mathbf{V}_m|^2)$, the benefit of a smaller computational requirement, obtained from using a small value of N, will be lost. In this case, without increasing the value of N, a linear interpolator and weighted averaging filter are used to increase the resolution of $\mathbf{H}(|\mathbf{V}_m|^2)$ and to improve the accuracy of the $\mathbf{H}(|\mathbf{V}_m|^2)$, respectively.

The linear interpolator achieves two major functions: constructing a piecewise continuous $\mathbf{H}(|\mathbf{V}_m|^2)$ function from the output of the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, and generating a new predistortion LUT. These functions can be formulated as:

$$\mathbf{H}\left(\left|\mathbf{V}_{m_j}\right|^2\right) = \mathbf{H}\left(\left|\hat{\mathbf{V}}_{m_i}\right|^2\right) + \frac{H\left(\left|\hat{\mathbf{V}}_{m_{i+1}}\right|^2\right) - H\left(\left|\hat{\mathbf{V}}_{m_i}\right|^2\right)}{\left|\hat{\mathbf{V}}_{m_{i+1}}\right|^2 - \left|\hat{\mathbf{V}}_{m_i}\right|^2} \cdot \left(\left|\mathbf{V}_{m_j}\right|^2 - \left|\hat{\mathbf{V}}_{m_i}\right|^2\right)$$

$$\left|\hat{\mathbf{V}}_{m_i}\right|^2 \le \left|\mathbf{V}_{m_j}\right|^2 < \left|\hat{\mathbf{V}}_{m_{i+1}}\right|^2 \qquad (3\text{-}11)$$

$$i \in [0 \cdots N-1] \quad \cap \quad j \in [0 \cdots M-1]$$

where N is the number of data points of the discontinuous output from the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator and M is the size of the LUT. Using the linear interpolator allows the linearizer to generate a LUT with M complex points from a relatively small (N) sample set; e.g., the linearizer may generate a 1024 point LUT from 64 $\mathbf{H}(|\mathbf{V}_m|^2)$ values. In this case, if the L/S ratio, i.e., the value of M/N, is large enough, the linear interpolator can reduce the computational requirement for generating a M complex point LUT when compared with only using the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator and M sampled sets of data.

Since the linear interpolator cannot suppress the effects of quantization noise, every output value from the linear interpolator is fed to a weighted averaging filter before it is used to update the LUT. The weighted averaging filter smoothes the piecewise continuous $\mathbf{H}(|\mathbf{V}_m|^2)$ function, and provides a good approximation to the $\mathbf{H}(|\mathbf{V}_m|^2)$ function. Eq. (3-12) defines the operation of the weighted averaging filter:

$$y(k)=\begin{cases}0 & k\le 0\\ a\cdot y(k-1)+b\cdot x & k=1,2,3,\cdots\end{cases} \tag{3-12}$$

where $a>b>0$, $a+b=1$, k is the time index, $y(k)$ is a value of $\mathbf{H}(|\mathbf{V}_m|^2)$ at time interval k and x is a new estimated value of $\mathbf{H}(|\mathbf{V}_m|^2)$ from the linear interpolator. As shown in (3-12), the weighted averaging filter is a first order causal IIR filter. The unit-sample response $h_w(k)$ and the z-transform of the filter's transfer function $\mathbf{H}_w(z)$ are given by (3-13):

$$\begin{aligned} h_w(k) &= b\cdot a^n\cdot u(k)\\ \mathbf{H}_w(z) &= \frac{b}{1-a\cdot z^{-1}} \end{aligned} \tag{3-13}$$

where $u(k)$ is the unit-step response. Because the coefficient "a" is less than unity, the pole of $\mathbf{H}_w(z)$ is located inside the z-plane unit circle. Therefore, the weighted averaging filter is unconditionally stable. According to (3-13), the weighted averaging filter is analogous to a RC lowpass filter with attenuation "b" and a time constant $\tau=-T_u/\ln(a)$, (where T_u is the time period for updating the LUT). The time constant (τ) is one of the important factors that determine the convergence time for obtaining a stable set of LUT values in the enhanced linearization method. Because the weighted averaging filter acts like a lowpass filter, high frequency noise on the latched signals is reduced. In addition, the weighted averaging filter also generates an averaged $\mathbf{H}(|\mathbf{V}_m|^2)$ function which can compensate for the discontinuous response from the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator and the linear interpolator.

3.5 Summary of this Chapter

The enhanced linearization method shown in Figure 3-1 consists of two processing units: the real-time predistortion unit and the off-line distortion estimation unit. The real-time predistortion unit generates a predistorted signal according to the quadrature input signal magnitude and the corresponding predistortion LUT I and Q values. The off-line distortion estimation unit, on the other hand, is used to generate the predistortion LUT according to three sampled signals ($\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$). This processing unit consists of five major components: a delay compensator, $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, mapping controller, linear interpolator and weighted averaging filter, which are characterized by Eqs. (3-9) to (3-12). Since these components are simple and unconditionally stable, a stable and relatively high bit rate linearizer can be implemented using a DSP. In order to enhance the functionality of the linearizer, two subsidiary loops, namely a clipping control loop and a power regulation control loop, are attached to the enhanced linearizer. The DSP implementation of the enhanced linearizer with the two subsidiary loops is described in detail in the next chapter. Experimental results and discussion of the implemented linearizer are presented in Chapter 5.

4. IMPLEMENTATION OF THE ENHANCED LINEARIZER

In this chapter, the implementation of a simplified digital transmitter with the enhanced linearizer is described. As shown in Figure 4-1, the simplified transmitter consists of seven parts: a data source, a π/4 DQPSK encoder and pulse shaping filter, the enhanced linearizer, a quadrature modulator and demodulator, and a RF power amplifier. The corresponding hardware requirement of each part is also shown in Figure 4-1. The following sections describe the detailed implementation of each part.

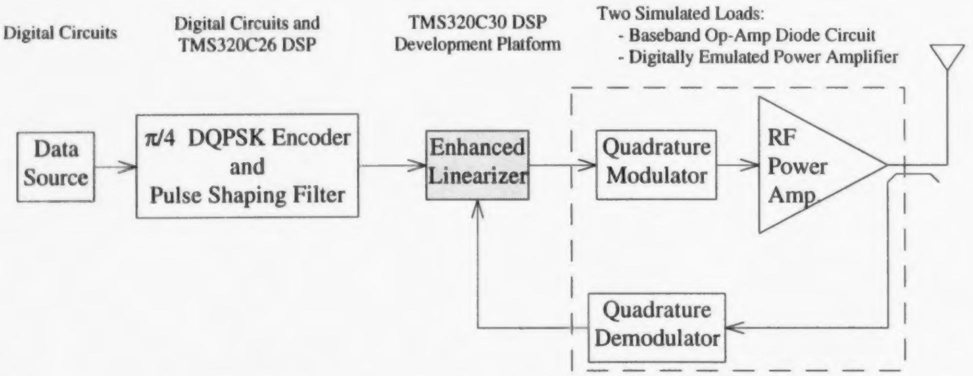

Digital Circuits
Digital Circuits and TMS320C26 DSP
TMS320C30 DSP Development Platform
Two Simulated Loads:
- Baseband Op-Amp Diode Circuit
- Digitally Emulated Power Amplifier
Data Source
π/4 DQPSK Encoder and Pulse Shaping Filter
Enhanced Linearizer
Quadrature Modulator
RF Power Amp
Quadrature Demodulator

Figure 4-1. Block diagram of a simplified digital transmitter with the enhanced linearizer.

4.1 Data Source

The data source provides a digital data stream to the rest of the transmitter. For voice transmission, the content of the data stream is a digitized voice signal, which has been compressed and framed into a specific format [1]. In this case, the content of the output data stream is almost random. Therefore, in the present implementation, the data source is a pseudorandom binary data generator. Two methods [31], namely the multiplicative congruential method or mixed congruential method, and the M-sequence method, are used for generating the pseudorandom binary data. Eqs. (4-1) and (4-2) give the mathematical definition of these methods. For the congruential method:

$$X_i = A \cdot X_{i-1} + C \bmod R \qquad i = 1,2,3,\cdots \qquad (4\text{-}1)$$

where i is a discrete time interval, X_i is the output data sequence, A, C and R are constants and $X_o>0$, $A>0$, $C\geq 0$, $R>X_o$, $R>A$, $R>C$, and *mod* is the modulo operator. For the M-sequence method:

$$d_i = \sum_{k=1}^{m} \alpha_k \cdot d_{i-k} \qquad i=1,2,3,\cdots \qquad (4\text{-}2)$$

where i is a discrete time interval, $d_i \in [0,1]$ is the output data sequence, m is the tap length, $\alpha_k \in [0,1]$ is the tap coefficients and $\sum$ is the modulo-2 summation.

As shown in Eq. (4-1), the multiplicative congruential method requires some mathematical calculations, i.e., multiplication, addition and modulo-R operation. It is more suitable for computer program applications. On the other hand, the M-sequence method requires modulo-2 summation; i.e., Exclusive-OR, and it only depends on the values of m, α_k and the initial value of d_k. It can be easily implemented using simple digital circuits. Therefore, in this project, the M-sequence method has been adopted to realize the data source. Since d_i is repeated after every 2^m-1 outputs, it is preferable to set the value of m as large as possible. Considering hardware limitations, m was set to a tap length of 17. Thus, if the bit rate of the data source is made 18.9 kbps, the output data sequence has a period of ~7.0 seconds, which is sufficient for the experimental testing. When the tap length is set, the minimum number of the non-zero tap coefficient can be obtained from a standard M-sequence coefficient table [31]. Eq. (4-3) shows the mathematical definition of the data source.

$$\alpha_k = \begin{cases} 1 & k = 3,7 \\ 0 & (1 \le k \le m) \cap (k \ne 3,7) \end{cases}$$

$$d_i = d_{i-3} \oplus d_{i-17} \qquad i = 0,1,2,\cdots \tag{4-3}$$

where $\oplus$ is modulo-2 addition. A schematic diagram of the implemented data source is shown in Figure 4-2, in which the data source consists of seventeen clocked registers and one exclusive-OR logic gate. These clocked registers form a 17 tap delay line, and the outputs of Reg 2 and Reg 16 are connected to the inputs of the exclusive-OR. Since the initial values set in the registers must not all be zero, the first register, Reg 0, in this case, is set to 1; the rest are set to zero in the initialization. In addition, because the output data from the data source is required to be synchronized to the system clock, the output of Reg 16 is used for the output of the data source, instead of the output of the exclusive-OR gate.

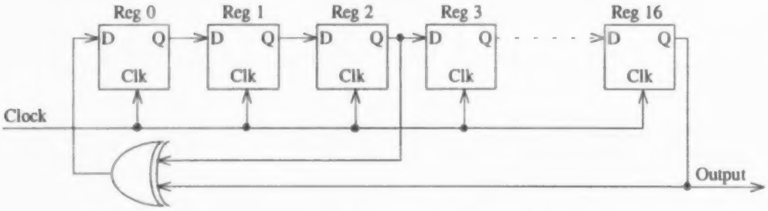


Figure 4-2. Diagram of the implemented data source.

The circuit diagram of the data source is shown in Appendix B, Figure B-1. In order to initialize each register to a specific value (Reg 0 = 1 and Reg 1- 16 = 0), a diode reset circuit, shown in the left-most part of the diagram, is used. This circuit initializes the registers when the power is turned on. Moreover, a simple RC delay circuit is also added to the connection between 74ALS74 and 74LS164, so that it can compensate for the time incompatibility between two kinds of TTL digital circuitry that is used, i.e., the ALS family and LS family.

4.2 π/4 DQPSK Encoder and Pulse Shaping Filter

As described in Section 2.1.3, the π/4 DQPSK encoder applies the Gray code format and the π/4 DQPSK modulation scheme to encode a serial input digital data, provided by the data source, to a parallel in-phase and quadrature phase (I and Q) signal. The output of the π/4 DQPSK encoder is then passed to the pulse shaping filter, which is a square-root-raised-cosine filter with a roll-off factor of 0.35, to define the spectral shape of the transmitted signal. The resulting output signal from the π/4 DQPSK encoder and the pulse shaping filter has a spectral efficiency of 1.62 bit/Hz when the input data rate is 48.6 kb/s. Figure 4-3 shows a structural diagram of the π/4 DQPSK encoder and pulse shaping filter that were built in this project. These two components (π/4 DQPSK encoder and pulse shaping filter) are implemented using digital circuits and a Texas Instruments TMS320C26 DSP [32]. The design and implementation of these components is described in the following two sections [33, 34, 35].

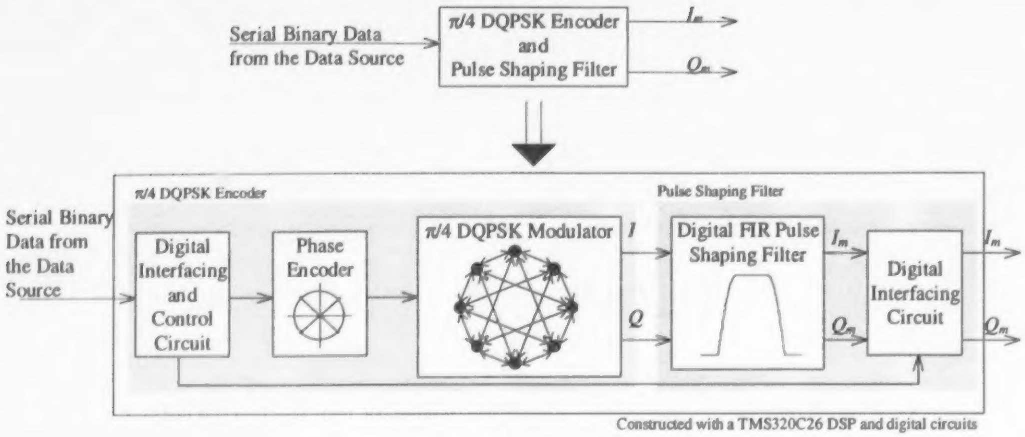

Serial Binary Data from the Data Source
π/4 DQPSK Encoder and Pulse Shaping Filter
I_m
Q_m
π/4 DQPSK Encoder
Pulse Shaping Filter
Serial Binary Data from the Data Source
Digital Interfacing and Control Circuit
Phase Encoder
π/4 DQPSK Modulator
I
Q
Digital FIR Pulse Shaping Filter
I_m
Q_m
Digital Interfacing Circuit
I_m
Q_m
Constructed with a TMS320C26 DSP and digital circuits

Figure 4-3. Structural diagram of the π/4 DQPSK encoder and pulse shaping filter.

4.2.1 Design and Implementation of the π/4 DQPSK Encoder

As shown in Figure 4-3, the π/4 DQPSK encoder consists of three components: a digital interfacing and control circuit, a phase encoder and a π/4 DQPSK modulator. The digital interfacing and control circuit is used to interface between the serial input data and the π/4 DQPSK encoder, and to generate several control signals to orchestrate the operation of the π/4 DQPSK encoder and pulse shaping filter. This component is implemented by digital circuit, and will be described in the following subsection. The phase encoder uses the Gray code format to convert two serial input data bits to a corresponding differential phase angle ($\Delta\phi$). Table 4-1 shows the relation between the input binary data and $\Delta\phi$.

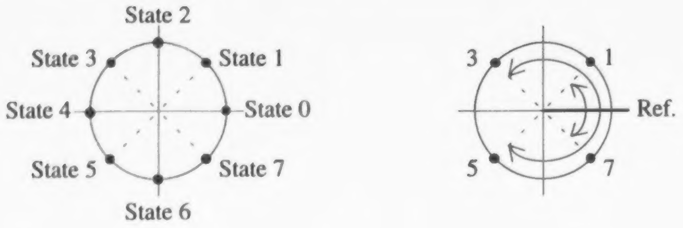
Table 4-1. RELATION BETWEEN THE SERIAL INPUT DATA BITS AND $\Delta\phi$.

Input Data (b_1, b_0)	Differential Phase Angle ($\Delta\phi$)
0,0	$+\pi/4 = +45^\circ$
0,1	$+3\pi/4 = +135^\circ$
1,1	$-3\pi/4 = -135^\circ$
1,0	$-\pi/4 = -45^\circ$

where b_0 and b_1 represent the current and the next input data bit, respectively. The output of the phase encoder is then fed to the π/4 DQPSK modulator in order to generate the I and Q signals. According to the IS-54 standard, the modulation scheme of the π/4 DQPSK modulator is defined as,

$$S_k = S_{k-1} \cdot e^{j \cdot \Delta\phi(b_{1_k}, b_{0_k})} \tag{4-4}$$

where k, S_k, and $\Delta\phi(b_{1_k},b_{0_k})$ are corresponding to the output symbol index, the k^{th} complex output symbol from the modulator, and the phase encoder function (see Table 4-1). b_{0_k} and b_{1_k} are the first and second input data bit from the data source. As shown in Eq. (4-4), S_k is generated by rotating the phase of the previous symbol S_{k-1} by an angle $\Delta\phi_s(b_{1_k},b_{0_k})$. Therefore, the output signal constellation from the π/4 DQPSK modulator has only eight possible output symbols which lie on a complex plane circle (see Figure 2-2). Because of this characteristic, Eq. (4-4) can be simplified by converting to a state equation. In this case, the eight output symbols can be denoted as eight states (i.e., State 0, 1,... ,7). Also, the $\Delta\phi(b_{1_k},b_{0_k})$ is transformed to a state function $\Delta\phi_s(b_{1_k},b_{0_k})$ by mapping the corresponding $\Delta\phi$ to these eight states with 0 degree, i.e., State 0, as the reference. As a result, the $\Delta\phi_s(b_{1_k},b_{0_k})$ provides four state values (i.e., 1, 3, 5, 7) to represent the corresponding differential phase angles. Figure 4-4 shows the constellation state diagram and $\Delta\phi_s(b_{1_k},b_{0_k})$ state diagram.



(a) (b)

Figure 4-4. (a) Constellation state diagram.
(b) $\Delta\phi_s(b1_k, b0_k)$ state diagram.

With these eight states and $\Delta\phi_s(b1_k, b0_k)$, the state equation of Eq. (4-4) can be written as:

$$i_k = \left(i_{k-1} + \Delta\phi_s\left(b1_k, b0_k\right)\right) \bmod 8 \qquad (4\text{-}5)$$

where $i \in [0, \ldots, 7]$ is the state number, and *mod* is the modulo operation. Since the DSP does not have any built-in modulo-8 operation, the equivalent function is realized using an AND instruction with a value of 7. Therefore, Eq. (4-5) is re-written as:

$$i_k = \left(i_{k-1} + \Delta\phi_s\left(b1_k, b0_k\right)\right) \text{ AND } 7 \qquad (4\text{-}6)$$

The actual corresponding output symbol can be obtained by looking up a state conversion table, which stores the complex values of the eight output symbols, indexed by *i*. As a result, the π/4 DQPSK modulator only requires a simple state equation Eq. (4-6) and a state conversion table. It is suitable for real-time implementation in the DSP.

The phase encoder state function $\Delta\phi_s(b1_k, b0_k)$ may also be implemented in the DSP by using another look-up table. On the other hand, it is possible to realize the $\Delta\phi_s(b1_k, b0_k)$ using digital circuit; resulting in a simpler implementation than using a table. As observed from Table 4-2, the input data bits have a very simple relationship, shown in Eq. (4-7), to the corresponding state values of $\Delta\phi_s(b1_k, b0_k)$ in binary form.

Table 4-2. RELATION BETWEEN THE SERIAL INPUT DATA BITS AND THE $\Delta\phi_s(b1_k, b0_k)$ FUNCTION.

Input Data (b_1, b_0)	$\Delta\phi_s(b1_k, b0_k)$ in decimal	$\Delta\phi_s(b1_k, b0_k)$ in binary (s_2,s_1,s_0)
0,0	1	0,0,1
0,1	3	0,1,1
1,1	5	1,0,1
1,0	7	1,1,1

$$\begin{aligned} s_0 &= 1 \\ s_1 &= b_0 \oplus b_1 \\ s_2 &= b_1 \end{aligned} \tag{4-7}$$

Using Eq. (4-7), $\Delta\phi_s(b1_k, b0_k)$ can be realized using one Exclusive-OR logic gate and some hardwiring. The next two subsections describe the software and hardware implementation for the π/4 DQPSK encoder.

4.2.1.1 Software Implementation

The software implementation is mainly used for realizing the π/4 DQPSK modulator. As shown in Eq. (4-6), the π/4 DQPSK modulator only consists of two calculations: addition and the AND operation. It is very simple to implement the modulator in the DSP. The pseudo code of the modulator's DSP program is shown below:

```
Get a new state value from the phase encoder;
Add the new state value to the previous state value;
Perform AND 7 operation on the resulting value;
Store the new calculated state value into memory;
Use the new calculated state value to look up  the corresponding complex
        symbol value from the state conversion table.
```

The implemented DSP program of the π/4 DQPSK modulator is shown in Appendix E, List E-1.

4.2.1.2 Hardware Implementation

There are three circuits, namely the phase encoder circuit, control circuit, and timing circuit, in the hardware implementation of the π/4 DQPSK encoder. These circuits are used to realize the digital interfacing and control circuit, and the phase encoder. They provide the functions of encoding the serial input data, interfacing to the TMS320C26 DSP, and generating control signals. Figure 4-5 shows a block diagram of these circuits.

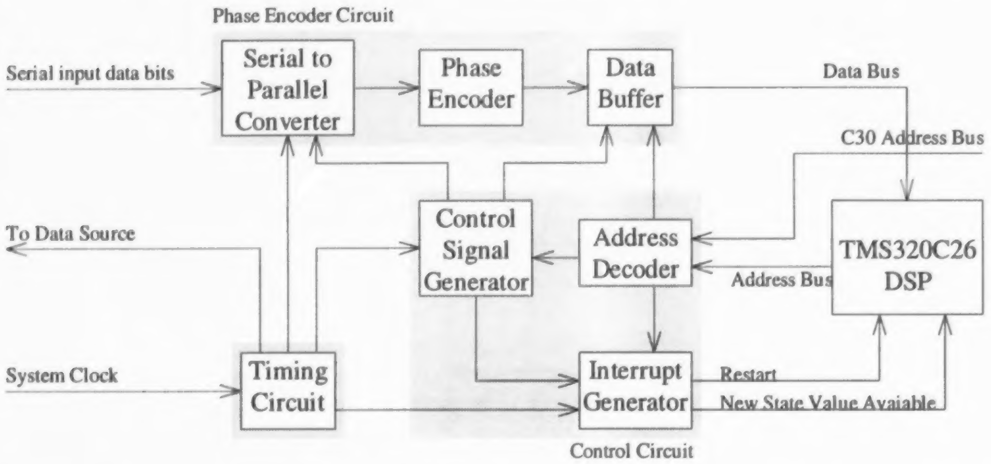

Phase Encoder Circuit
Serial input data bits
Serial to Parallel Converter
Phase Encoder
Data Buffer
Data Bus
C30 Address Bus
To Data Source
Control Signal Generator
Address Decoder
Address Bus
TMS320C26 DSP
System Clock
Timing Circuit
Interrupt Generator
Restart
New State Value Avaiable
Control Circuit

Figure 4-5. Block diagram of the hardware implementation of the π/4 DQPSK encoder

The phase encoder circuit consists of three parts: a serial-to-parallel converter, phase encoder, and data buffer. The serial-to-parallel converter is used to convert the serial input binary data, generated by the data source, to 2-bit binary data. The phase encoder then latches the 2-bit binary data, and applies Eq. (4-7) to encode the input data to the corresponding state value (see Table 4-2). Since the serial-to-parallel converter and state encoder are operated asynchronously to the DSP, a data buffer is used to store the encoded state value, and to synchronize the DSP's access. The control circuit, shown in Figure 4-5, consists of three parts: an address decoder, interrupt generator, and control signal generator. According to the values of the address buses in the TMS320C26 DSP and the TMS320C30 DSP, the address decoder generates an enable signal to activate one or more components, such as the control signal generator, the interrupt generator, and the data buffer. Since the TMS320C30 DSP is used to realize the linearizer, it generates some control signals to the π/4 DQPSK encoder and pulse shaping filter in order to synchronize the operation. The control signal generator, depending on the enable signal and the timing signal from the timing circuit, generates several control signals to the interrupt generator, the serial-to-parallel converter, and the data buffer in order to control the operation state of these circuits. The operation state includes the start/stop of the entire phase encoder circuit, the assert/release of the interrupt signal, and the activate/de-activate of the data buffer. The interrupt generator simply generates two interrupt signals to the TMS320C26 DSP when the entire π/4 DQPSK encoder circuit needs to restart, or when a new state value is available in the data buffer. In response to the interrupt signals, the TMS320C26 DSP will either restart the entire DSP program or load the new state value for computing a new set of output values. The timing circuit is used to modify the duty cycle of the system clock, and to generate several timing signals to synchronize the control circuit, the phase encoder circuit as well as the data source. Figures B-2 and B-3 in Appendix B shows the actual circuit diagrams of the phase encoder circuit, the control circuit, and the timing circuit.

4.2.2 Design and Implementation of the Pulse Shaping Filter

A pulse shaping filter is used to define the spectral shape of the $\pi/4$ DQPSK modulated signals (I and Q). According to the IS-54 standard, the pulse shaping filter is a square-root-raised-cosine filter with roll-off factor $\alpha=0.35$. Eq. (4-8) shows the frequency domain definition of this filter [1]:

$$|H(f)| \equiv \begin{cases} 1 & 0 \le |f| \le \dfrac{1-\alpha}{2\cdot \mathrm{T}} \\ \sqrt{\dfrac{1}{2}\cdot\left\{1-\sin\left[\dfrac{\pi\cdot(2\cdot f\cdot T-1)}{2\cdot\alpha}\right]\right\}} & \dfrac{1-\alpha}{2\cdot \mathrm{T}} \le |f| \le \dfrac{1+\alpha}{2\cdot \mathrm{T}} \\ 0 & |f| \le \dfrac{1+\alpha}{2\cdot \mathrm{T}} \end{cases} \qquad (4\text{-}8)$$

where T is the symbol period. Either an analog filter or digital filter can be used to realize this pulse shaping filter. When considering the factors of accuracy, flexibility and implementation time, the digital filter is the appropriate choice in this project. There are two basic types of digital filter: finite impulse response (FIR) filter and infinite impulse response (IIR) filter. Since the FIR filter has the advantages of stability, linear phase response, low round-off noise, and less ringing on glitches [36, 37], it is suitable for implementing the digital pulse shaping filter. Figure 4-6 shows a structure diagram of a standard FIR filter. A FIR filter consists of a N tap discrete delay line. The most recent N sampled data are stored in the delay line, and are multiplied by their corresponding tap coefficients. The products are then summed together to form the filter's output. The characteristic of the FIR filter is determined by the tap coefficients (h_i), and the length of the delay line (N).

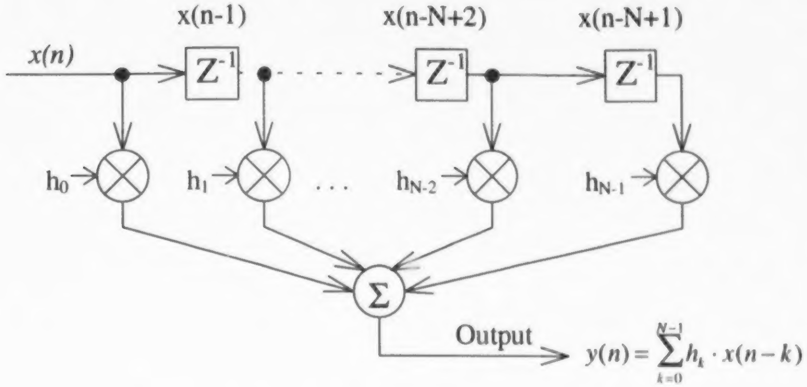


Figure 4-6. Block diagram of a standard FIR filter.

The values of h_i for the pulse shaping filter are obtained by digitizing the impulse response of the pulse shaping filter at the sampling rate. Eq. (4-9) and Figure 4-7 show the equation of the impulse response of the defined pulse shaping filter [2], and an example of obtaining h_i at the sampling rate (f_s) and N=25, respectively.

$$h(t) \equiv \begin{cases} 1-\alpha+4\cdot\dfrac{\alpha}{\pi} & t=0 \\[2ex] \dfrac{\alpha}{\sqrt{2}}\cdot\left[\left(1+\dfrac{2}{\pi}\right)\cdot\sin\left(\dfrac{\pi}{4\cdot\alpha}\right)+\left(1-\dfrac{2}{\pi}\right)\cdot\cos\left(\dfrac{\pi}{4\cdot\alpha}\right)\right] & t=\pm\dfrac{T}{4\cdot\alpha} \\[2ex] \dfrac{\sin\left[\pi\cdot(1-\alpha)\cdot\dfrac{t}{T}\right]+4\cdot\alpha\cdot\dfrac{t}{T}\cdot\cos\left[\pi\cdot(1+\alpha)\cdot\dfrac{t}{T}\right]}{\alpha\cdot\dfrac{t}{T}\cdot\left[1-\left(4\cdot\alpha\cdot\dfrac{t}{T}\right)^2\right]} & \text{for all other } t \end{cases} \quad (4\text{-}9)$$

where T is the symbol period.

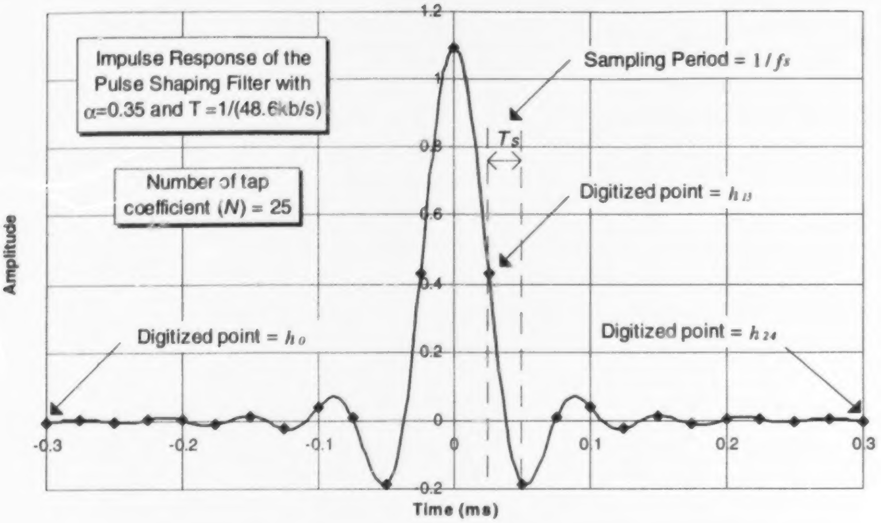


Figure 4-7. Example of obtaining h_i at the sampling rate (f_s) and N=25.

However, the value of N is difficult to be obtained analytically. In addition, the implementation structure of a FIR filter and the sampling rate also affect the value of N. There are two types of structure, which can be used to implement a FIR filter: a single rate structure and a multirate structure. The single rate structure requires only one standard N-tap FIR filter, which is operated at the sampling rate (f_s). This kind of structure is easy to implement, but it has the problem of requiring a large value of N when the value of f_s is larger than the data rate of the input signal. On the other hand, the multirate structure consists of two components: a simple FIR filter, and an interpolating filter. The simple FIR filter is operated at one M^{th} of f_s, where M is the oversampling rate with respect to one output symbol. The interpolating filter is used to interpolate the output from the simple FIR filter, and to generate an output with a sampling rate of f_s. Since the operating rate of a simple FIR filter is M times slower than f_s, the tap length of this FIR filter is relatively shorter than of the filter using the single rate structure. The interpolating filter may be implemented by using linear interpolator. In this case, the multirate structure requires less computation even though its implementation is not as straight forward. Therefore, in this project, the FIR pulse shaping filter is implemented using the multirate structure. In order to increase the accuracy, the interpolating filter interpolates the output from the simple FIR filter along the defined pulse shape. Figure 4-8 illustrates an output example of the interpolating filter. In this case, the interpolating filter can be implemented by using $(M-1)$ FIR filters; each FIR filter has the transfer characteristic of the defined pulse shaping filter and is delayed by $1/f_s$ from the previous filter. For the simplification, the simple FIR filter and the interpolating filter are realized using the same tap length of the FIR filter. As a result, the implemented pulse shaping filter finally consists of M N-tap FIR filters, which have similar transfer characteristics, but with different tap coefficients.

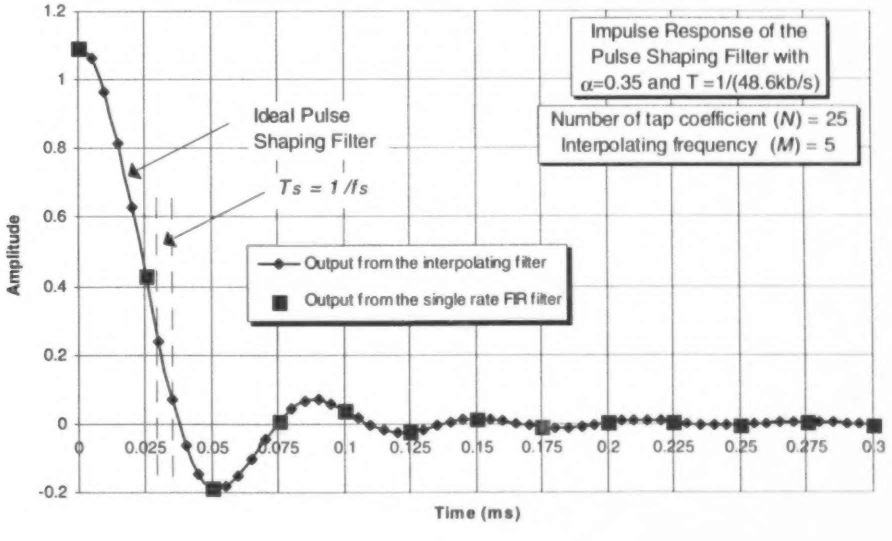


Figure 4-8. Example of the output from the interpolating filter.

According to the designed filter structure, a simulation program is written for determining the value of N and M. If the averaged numerical conversion error and the

According to the designed filter structure, a simulation program is written for determining the value of N and M. If the averaged numerical conversion error and the RMS error vector magnitude ($V_{err_{rms}}$) are restricted to be <0.1% and <5%, respectively, the value of N is found to be 8; and $M > 10$. In the simulation, the averaged numerical conversion error is defined as the averaged absolute percentage error of converting a floating point number to a 16 bit integer number. The $V_{err_{rms}}$ is defined as:

$$V_{err_{rms}} = \sqrt{\frac{\sum_{i=0}^{L} \left|\mathbf{V}_{err_i}\right|^2}{L}} \qquad (4\text{-}10)$$

where L is the number of measured points, and $\mathbf{V}_{err}$ is the error vector which is shown in Figure 4-9, and is defined as the difference between the reference pulse-shaped output vector ($\mathbf{V}_{\mathrm{ref}}$) and the measured output vector ($\mathbf{V}_{\mathrm{measured}}$). The $\mathbf{V}_{\mathrm{ref}}$ and $\mathbf{V}_{\mathrm{measured}}$ are generated by a 1024 tap FIR pulse shaping filter with floating point numbers, and a M N-tap FIR pulse shaping filter with 16 bit integer number, respectively.

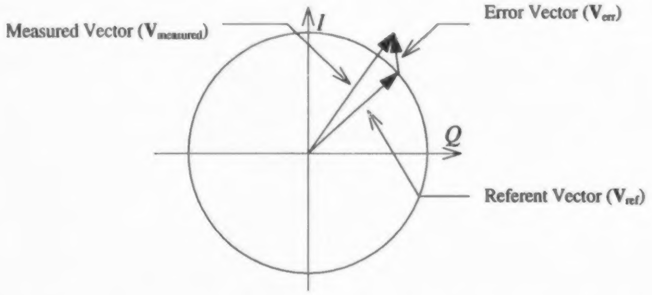

Measured Vector ($\mathbf{V}_{\mathrm{measured}}$)
I
Error Vector ($\mathbf{V}_{\mathrm{err}}$)
Q
Referent Vector ($\mathbf{V}_{\mathrm{ref}}$)

Figure 4-9. Representation of the error vector of the pulse shaped output.

Besides determining the values of N, M, and h_i, the implementation method needs to be considered. There are two common methods to implement a digital FIR filter: the Read-Only-Memory (ROM) method [33, 34] and the DSP method. The ROM method uses the input data as an index to locate a corresponding filter's output value from a ROM. All possible filter output values are pre-calculated and stored in the ROM beforehand. Using this method, a high speed digital FIR filter is easily achieved. However, the drawback of this method is that it requires a large size of ROM. On the other hand, the DSP method uses a digital signal processor to directly calculate the filter's output in real time. Although the speed of this type of filter cannot match to one which is implemented by the ROM method, it requires a much smaller memory and less circuit layout. When the flexibility and compatibility with the π/4 DQPSK encoder are considered, the DSP method is the better choice to be adopted for implementing the FIR pulse shaping filter in this project. In the next two subsections, the software and hardware implementation of the filter are described.

4.2.2.1 Software Implementation

Since the π/4 DQPSK encoder generates a vector signal, i.e., I and Q signals, two digital FIR pulse shaping filters, based on the previous discussion, are implemented using a Texas Instrument TMS320C26 DSP. These two filters are assembled by M individual pairs of standard N-tap FIR filter. To realize this structure and to facilitate the DSP program development, the object-oriented programming (OOP) technique [38, 39] is selected in the filter implementation. The DSP program of the pulse shaping filters consists of M filter objects. Each object represents two N-tap FIR filters: one for the I signal, another for the Q signal. The calculated results, i.e., the I and Q signal, are sequentially sent to output buffers, which are described in the next section. In order to link these M filter objects together, two operating modes are used: the burst mode and the synchronous mode. In the burst operating mode, the first filter object in the link operates synchronously at one M^{th} of the sampling clock. The rest of the object works sequentially after the first one. Thus, M pairs of I and Q data output are obtained from the pulse shaping filter whenever a new symbol is received from the output of the π/4 DQPSK encoder. Using the burst operating mode, the maximum input data rate of the filter achieved is up to 43.5 kb/s if N=8 and M=16. The drawback of using this operating mode is that a pair of dynamic buffers with length M is required on the filter's output in order to synchronize the sampling clock, resulting in a more complicated hardware design. On the other hand, the synchronous operating mode requires a very simple buffer circuit on the filter's output, because each filter object operates synchronously to the sampling clock. However, in the synchronous operating mode, each filter object wastes most of its time waiting for a synchronized signal, and acknowledging this signal. As a result, the maximum input bit rate of the pulse shaping filter is only 20 kb/s, which is 54% less than the one operated in the burst mode. Considering the implementation time and hardware factors, the synchronous operating mode is good for realizing the digital FIR pulse shaping filter in this project.

Up to now, the π/4 DQPSK modulator and pulse shaping filter have been separately implemented into the DSP program. In order to optimize the entire DSP program, the modulator's program is broken down into seven different parts. Each part is then embedded into the corresponding filter objects in the pulse shaping filter. In this case, the modulator's program shares some of the waiting time in each filter object, resulting in a 25% process improvement in the entire DSP program. The maximum input bit rate now is ~25 kb/s in the synchronous operating mode; i.e., the maximum sampling rate is equal to 0.5 symbol/bit·25 kb/s·16 sample/symbol = 200 ksample/s. Figure 4-10 shows the software architecture of the entire implemented DSP program for both the π/4 DQPSK modulator and the pulse shaping filter. The DSP program list is shown in Appendix E, List E-1.

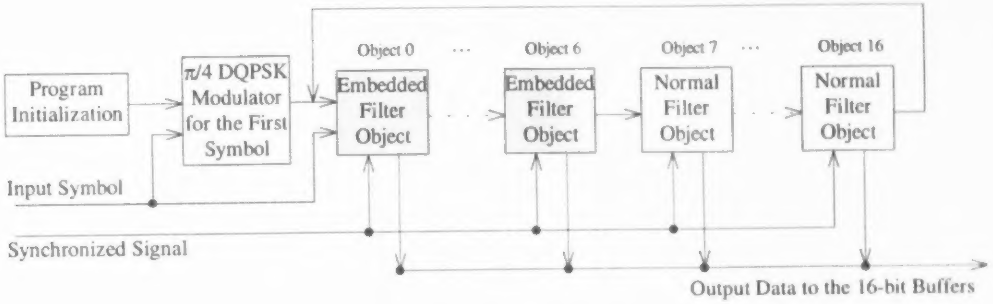

Program Initialization
π/4 DQPSK Modulator for the First Symbol
Input Symbol
Synchronized Signal
Object 0
Embedded Filter Object
Object 6
Embedded Filter Object
Object 7
Normal Filter Object
Object 16
Normal Filter Object
Output Data to the 16-bit Buffers

Figure 4-10. Software architecture of the entire implemented DSP program

Although the configuration of the implemented digital FIR pulse shaping filter, shown in Figure 4-10, is set to N=8 and M=16, the maximum input bit rate can be further increased by reducing the value of M, but not below 10. For example, the maximum input bit rate is increased by a factor of 1.6 when the value of M is reduced by 6. In this case, the new pulse shaping filter can be easily realized by removing six of the normal filter objects from the originally implemented DSP program, and re-calculating the tap coefficients.

4.2.2.2 Hardware Implementation

Although the pulse shaping filter is largely realized by a DSP program, a interfacing circuit is still required to buffer the output of the filter. As described in the previous section, the design of the interfacing circuit depends on the operating mode used in the DSP program. Since the synchronous operating mode is adopted for implementing the digital pulse shaping filter, the interfacing circuit is realized using a pair of 16-bit buffers. The block diagram and circuit diagram of the interfacing circuit for the digital pulse shaping filter are shown in Figure 4-11 and Appendix B, Figure B-3, respectively.

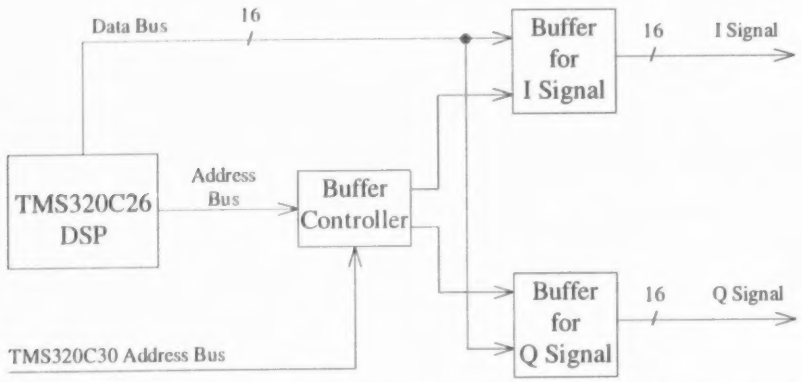


Figure 4-11. Block diagram of the interfacing circuit for the digital pulse shaping filter.

As shown in Figure 4-11, the interfacing circuit consists of two components: two 16-bit buffers for the I and Q signal, as well as buffer controller. The two 16-bit buffers provide a data access interface between the TMS320C26 DSP and the external device(s). The buffer controller monitors the TMS320C26 DSP address bus and the TMS320C30 DSP address bus, and generates appropriate control signals to the 16-bit buffers. These control signals are used to signal the buffers to latch a new set of filter output, or to enable the buffers for external access.

4.3 Enhanced Linearizer

As described in Chapter 3, the enhanced linearizer consists of two major processing units: a real-time predistortion unit and an off-line distortion estimation unit. The real-time predistortion unit is primarily used to generate a predistortion signal ($\mathbf{V}_p$). The off-line distortion estimation unit is used to sample the input signals, and to generate a predistortion LUT. These processing units, cooperating with ADC/DAC, quadrature modulator, RF power amplifier and quadrature demodulator, form a linearization loop to linearize the power amplifier. Besides the major processing units, the enhanced linearizer consists of two additional processing units, namely power regulation unit and the clipping control unit. These additional units are attached the linearization loop to form two other control loops: the power regulation control loop and the clipping control loop. The aim of these control loops is to control the transmitted signal power or to limit the $\mathbf{V}_p$ signal level. A loop selector is used to monitor the operation of these two control loops and to determine which loop should be activated. Figure 4-12 shows a simplified block diagram of the entire enhanced linearizer.

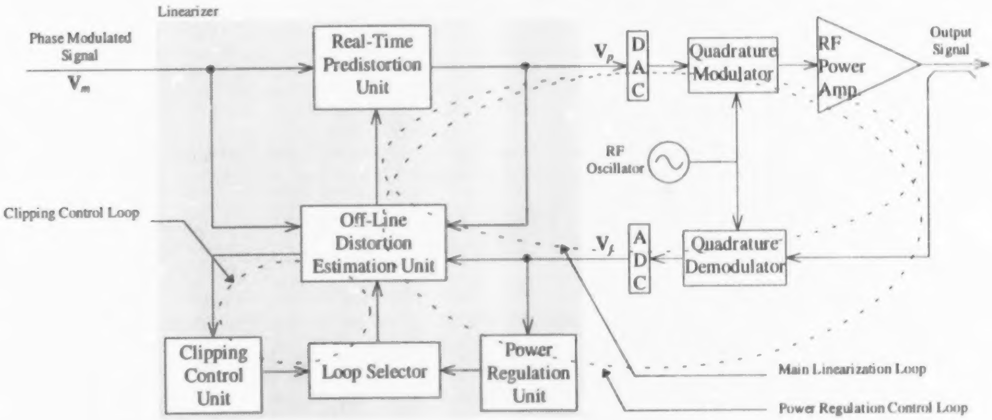

Linearizer
Phase Modulated Signal
$\mathbf{V}_m$
Real-Time Predistortion Unit
$\mathbf{V}_p$
D A C
Quadrature Modulator
RF Power Amp
Output Signal
RF Oscillator
Clipping Control Loop
Off-Line Distortion Estimation Unit
$\mathbf{V}_f$
A D C
Quadrature Demodulator
Clipping Control Unit
Loop Selector
Power Regulation Unit
Main Linearization Loop
Power Regulation Control Loop

Figure 4-12. Simplified block diagram of the entire enhanced linearizer.

In the original plan, the enhanced linearizer was to be completely implemented using a Texas Instruments TMS320C30 DSP development platform with dual 16 bits ADC/DAC ports. However, because of time constraints on the project, part of the linearization algorithm was implemented in an IBM PC 486 compatible computer. Since the linearizer was constructed in the TMS320C30 DSP development platform and the 486 computer, no particular hardware needed to be designed for the linearizer; all of the required hardware was built in the development platform [40]. In this case, hardware implementation of the enhanced linearizer was unnecessary. On the other hand, most of the effort has been expended on software implementation of the linearizer. Figure 4-13 shows a detailed functional block diagram of the enhanced linearizer, in which five shadowed boxes correspond to the units inside the linearizer. The following sections elaborate the content of these units. After that, the software architecture of the entire enhanced linearizer is described.

4.3.1 Real-Time Predistortion Unit

The real-time predistortion unit consists of three components: the $|\mathbf{V}_m|^2$ computation program, vector multiplier, and predistortion LUT. The $|\mathbf{V}_m|^2$ computation program calculates the square of the instantaneous magnitude of the input quadrature modulated signal $|\mathbf{V}_m|^2$. The calculated value of $|\mathbf{V}_m|^2$ then indexes to the predistortion LUT and locates a corresponding complex predistortion value. The vector multiplier performs a vector multiplication for the input $\mathbf{V}_m$ and the corresponding predistortion value, resulting in the predistorted signal $\mathbf{V}_p$.

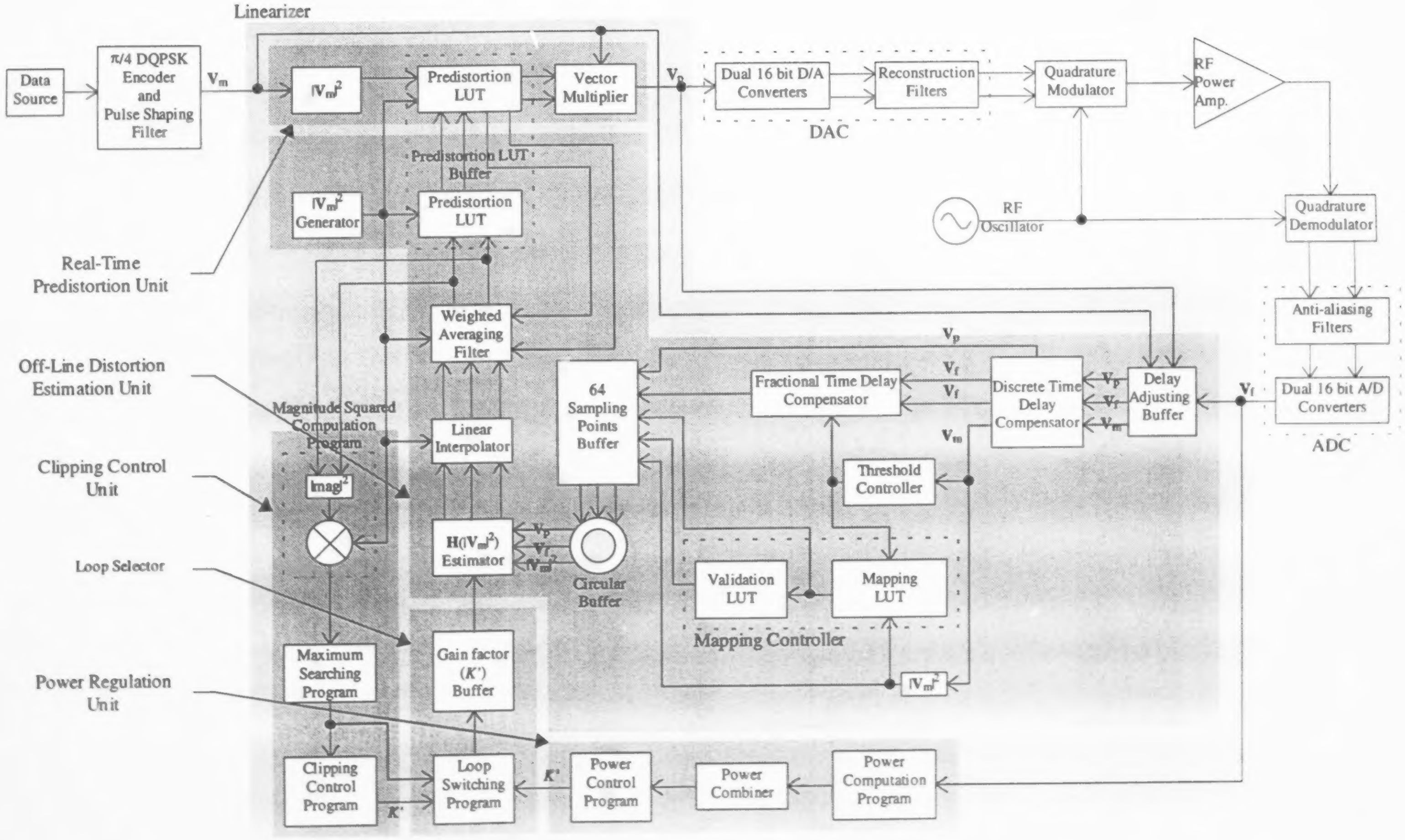

Linearizer
Data Source
π/4 DQPSK Encoder and Pulse Shaping Filter
V_m
$|V_m|^2$
Predistortion LUT
Vector Multiplier
V_p
Dual 16 bit D/A Converters
Reconstruction Filters
DAC
Quadrature Modulator
RF Power Amp.
RF Oscillator
Quadrature Demodulator
Anti-aliasing Filters
Dual 16 bit A/D Converters
ADC
V_f
Predistortion LUT Buffer
$|V_m|^2$ Generator
Predistortion LUT
Real-Time Predistortion Unit
Weighted Averaging Filter
Off-Line Distortion Estimation Unit
Magnitude Squared Computation Program
Linear Interpolator
64 Sampling Points Buffer
V_p
V_f
V_f
V_m
Fractional Time Delay Compensator
Discrete Time Delay Compensator
Delay Adjusting Buffer
V_p
V_f
V_m
Clipping Control Unit
$|\mathrm{mag}|^2$
Threshold Controller
$H(|V_m|^2)$ Estimator
V_p
V_f
$|V_m|^2$
Circular Buffer
Validation LUT
Mapping LUT
Mapping Controller
Loop Selector
Maximum Searching Program
Gain factor (K') Buffer
$|V_m|^2$
Power Regulation Unit
Clipping Control Program
K'
Loop Switching Program
K'
Power Control Program
Power Combiner
Power Computation Program

Figure 4-13. Detailed functional block diagram of the enhanced linearizer.

The predistortion LUT is one of the two predistortion LUTs in the predistortion LUT buffer. It stores the complex values of the predistortion function $\mathbf{H}(|\mathbf{V}_m|^2)$, and is indexed by the value of $|\mathbf{V}_m|^2$. The content of this predistortion LUT is shared by the real-time predistortion unit and the off-line distortion estimation unit, and is used to produce the predistorted signal or a new predistortion LUT. In order to cover the entire value range of $|\mathbf{V}_m|^2$ and to provide enough resolution to represent the $\mathbf{H}(|\mathbf{V}_m|^2)$, the predistortion LUT contains 1024 complex points, and occupies 2×1024 word memory, where each word has 32 bits. The size of the predistortion LUT is determined by a simulation, written by T. Neufeld [41]. Figure 4-14 shows the simulation result of the decreasing out-of-band power spectral density (psd) of the amplified signal, as the size of the LUT is increased.

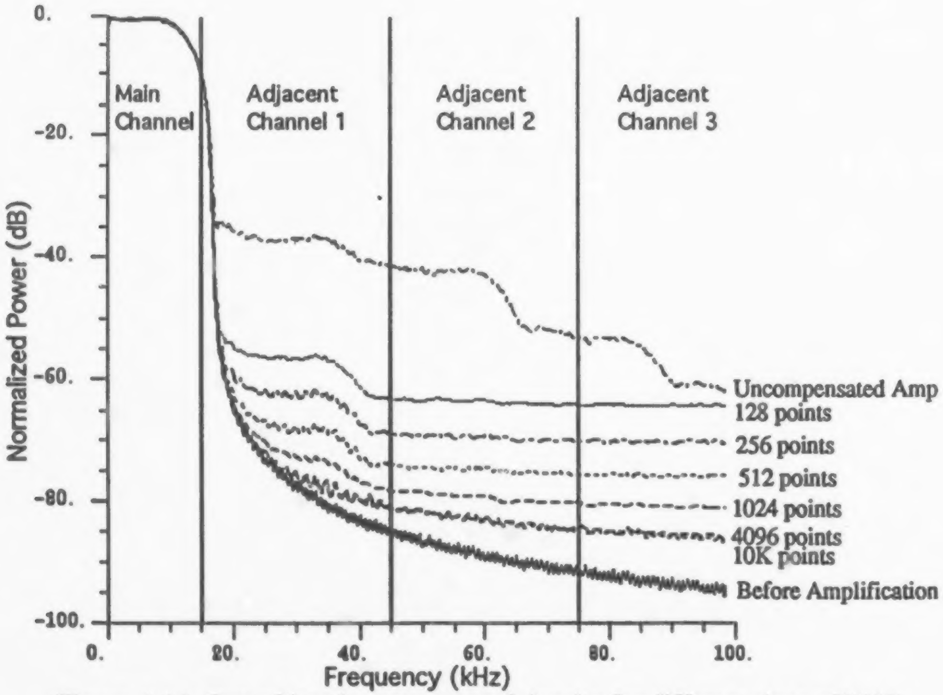


Figure 4-14. Out-of-band power spectral density for different sizes of LUT.

As shown in the simulation, a 1024 point predistortion LUT provides ~5 dB of out-of-band psd degradation from the ideal amplified output in adjacent channel 1, and a relatively stable -80 dB out-of-band psd in the second and third adjacent channels. Moreover, a 1024 point predistortion LUT is more practical for implementation than one having 4096 or 10,000 points.

4.3.2 Off-Line Distortion Estimation Unit

The off-line distortion estimation unit is the most complicated unit in the linearizer. It consists of four buffers and nine software programs. The delay adjusting buffer is the input front end buffer in the linearizer. It loads the sampled complex signals ($\mathbf{V}_m$, $\mathbf{V}_p$, and $\mathbf{V}_f$) from the $\pi/4$ DQPSK encoder and pulse shaping filter, the real-time predistortion unit, as well as the A/D converters, respectively, and stores them into the buffer's memory. In conjunction with the discrete time delay compensator and the fractional time delay compensator, this buffer corrects the time delay in $\mathbf{V}_f$. Every new set of samples is loaded into the buffer's memory, located behind the previous set of samples. If the buffer has been filled, no further sampled data are saved in the buffer until the fractional time delay compensator resets the buffer. The set number of the sampled signals saved in the buffer is equal to the buffer length, which is determined by the discrete time delay (τ_d) in $\mathbf{V}_f$. The value of τ_d is obtained from the phase characteristic of the anti-aliasing filter and the reconstruction filter in the ADC/DAC. Eq. (3-1), defined in Section 3.1, is used to estimate the value of τ_d. Figure 4-15, provided by [40], shows the group delay and characteristics of the anti-aliasing filter and the reconstruction filter in the ADC/DAC.

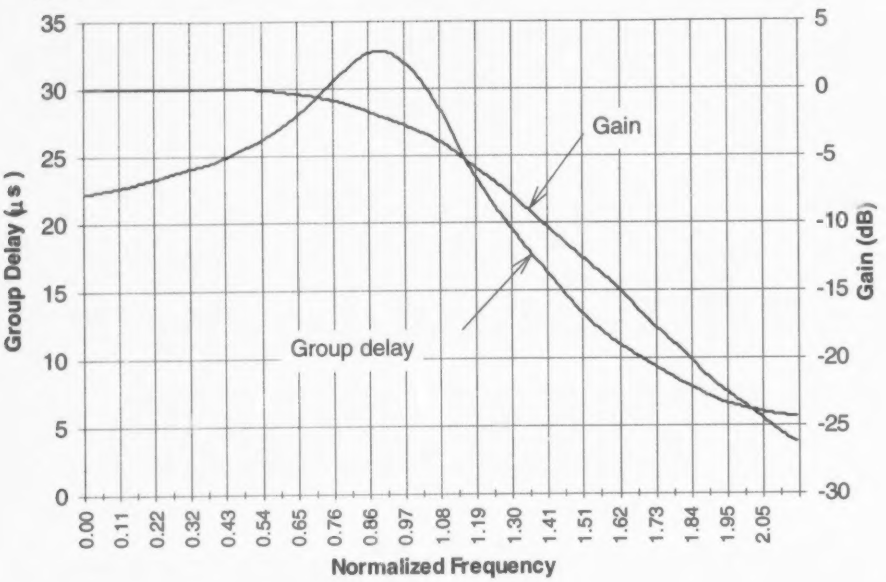


Figure 4-15. Characteristics of the anti-aliasing filter and the reconstruction filter in the ADC/DAC.

If the cut-off frequency of these filters and of the input data rate are set to be ~31 kHz and 18.9 kb/s, respectively, then the averaged group delay of each filter (τ_{filter}) is ~23 μs. When the sampling rate is set to be 151.2 ksample/s, then τ_d in terms of the sampling cycle can be obtained from the following calculation:

$$\begin{aligned}\tau_d &= RND\left(\frac{2\times 23\mu\text{s}}{1/151.2\text{kHz}}\right)\\ &= 7\text{ sampling cycles}\end{aligned} \tag{4-11}$$

Since the fractional time delay compensator requires one more set of sampled data for the linear interpolation, one additional sampling cycle is required. Therefore, the actual length of the delay adjusting buffer is equal to (7+1)+1 = 9; i.e., 9 continuous sets of the sampled $\mathbf{V}_m$, $\mathbf{V}_p$, and $\mathbf{V}_f$ signals are stored in the delay adjusting buffer, and provide 8 sampling cycles of delay. The memory size of this buffer is 6×9 words, where 6 is the total number of words required to represent the complex data $\mathbf{V}_m$, $\mathbf{V}_p$, and $\mathbf{V}_f$.

The discrete time delay compensator has the simplest software program in the off-line predistortion estimation unit. It cooperates with the delay adjusting buffer to correct the discrete time delay in $\mathbf{V}_f$. Because the discrete time delay in $\mathbf{V}_f$ is equal to 7 sampling cycles, this program simply accesses sets 1 and 8 of the stored samples in the delay adjusting buffer, and retrieves the corresponding values of $\mathbf{V}_m$, $\mathbf{V}_p$ and $\mathbf{V}_f$. As a result, the discrete time delay in $\mathbf{V}_f$ is corrected. Since the fractional time delay compensator requires an additional set of samples for correcting the fractional time delay, the discrete time compensator also retrieves the value of $\mathbf{V}_f$ from the delay adjusting buffer at the position of set 9. The discrete time delay compensator accordingly passes the retrieved $\mathbf{V}_m$, $\mathbf{V}_p$ and two $\mathbf{V}_f$ data to the 64-sampling-point buffer and three other software programs, namely the $|\mathbf{V}_m|^2$ computation program, the threshold controller, and the fractional time delay compensator. Figure 4-16 shows how the discrete time delay compensator works.

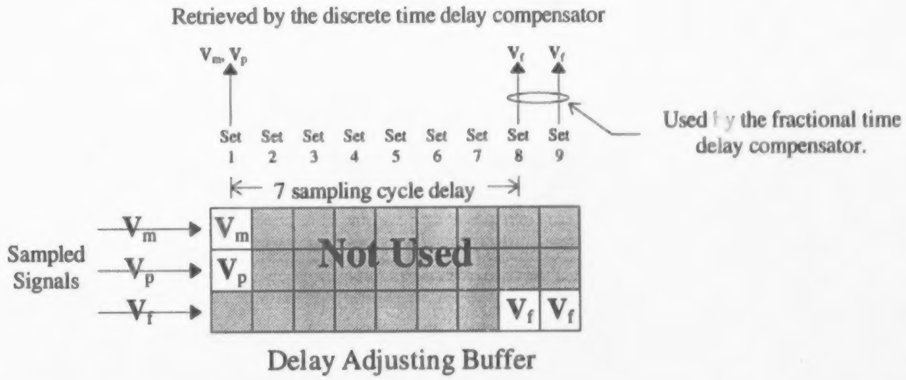

Retrieved by the discrete time delay compensator
V_m, V_p
V_f
V_f
Used by the fractional time delay compensator.
Set 1
Set 2
Set 3
Set 4
Set 5
Set 6
Set 7
Set 8
Set 9
7 sampling cycle delay
Sampled Signals
V_m
V_p
V_f
V_m
V_p
Not Used
V_f
V_f
Delay Adjusting Buffer

Figure 4-16. Operation of the discrete time delay compensator.

The $|\mathbf{V}_m|^2$ computation program is used to calculate the magnitude squared of the sampled value of $\mathbf{V}_m$, obtained from the discrete time delay compensator. The result is output to the 64-sampling-point buffer and the mapping controller. The threshold controller is used to detect the signal level of the $\mathbf{V}_m$. If the amplitude of the quadrature components (I_m and Q_m) of $\mathbf{V}_m$ is below the reference signal level, the threshold controller generates a disable signal to the mapping controller and the fractional time delay compensator in order to block the inputs to these two programs. As a result, the variation of the estimated predistortion values, generated by the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, can be eliminated when I_m and Q_m are below the reference. The value of the reference signal level is obtained experimentally, and is set to 750, which is equivalent to 68 mV.

The fractional time delay compensator uses the linear interpolation to correct the fractional time delay in the quadrature components (I_f and Q_f) of the $\mathbf{V}_f$. Since the quadrature modulator and demodulator are assumed to be perfect, the effect of the fractional time delay is contributed mainly by the anti-aliasing filter and the reconstruction filter in the ADC/DAC. The fractional time delay compensator receives two values of $\mathbf{V}_f$ from the discrete time delay compensator, and applies Eq. (3-4), given in Section 3.1, to the received $\mathbf{V}_f$ data in order to correct the fractional time delay in I_f and Q_f. The corrected I_f and Q_f are then passed to the 64-sampling-point buffer. Besides performing the correction in $\mathbf{V}_f$, the fractional time delay compensator generates a reset signal to the delay adjusting buffer whenever the entire correction process is finished, or whenever a disable signal, generated by the threshold controller, is received. This reset signal is used to activate the delay adjusting buffer to store new sampled signals.

The fractional time delay coefficients τ_I and τ_Q, shown in Eq. (3-4), are calculated by the calibration program before the linearization process starts. Since the distortion of the power amplifier is not included in the calculation of the τ_I and τ_Q, a pair of switches is used to bypass the quadrature modulator, power amplifier, and quadrature demodulator; i.e., the output I and Q signal of the reconstruction filters in the DAC are shorted to the input I and Q signal of the anti-aliasing filters in the ADC. During the calibration process, the calibration program takes over the entire linearizer, and directly outputs the $\mathbf{V}_m$ signal from the π/4 DQPSK encoder and pulse shaping filter to the DAC. The feedback complex signal is then digitized by the ADC. The calibration program continuously samples $\mathbf{V}_p$ and $\mathbf{V}_f$, and applies the sampled data to Eq. (3-3), defined in Section 3.1, to iterate the value of the τ_I and τ_Q. The iteration is stopped when the variation of τ_I and τ_Q is less 0.2%. Figure 4-17 shows a block diagram of the calibration. The program listing for the calibration program is shown in Appendix E, List 8.

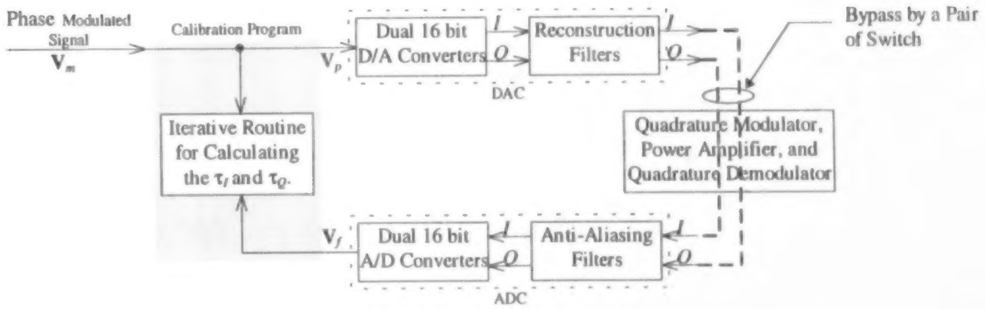

Phase Modulated Signal
$\mathbf{V}_m$
Calibration Program
$\mathbf{V}_p$
Dual 16 bit D/A Converters
I
Q
Reconstruction Filters
I
Q
DAC
Bypass by a Pair of Switch
Iterative Routine for Calculating the τ_I and τ_Q.
Quadrature Modulator, Power Amplifier, and Quadrature Demodulator
$\mathbf{V}_f$
Dual 16 bit A/D Converters
I
Q
Anti-Aliasing Filters
I
Q
ADC

Figure 4-17. Block diagram of the delay calibration process.

The mapping controller is used to select appropriate sets of sampled signals for calculating the values of $\mathbf{H}(|\mathbf{V}_m|^2)$. The characteristic of this controller was described in Section 3.3. According to the mapping transfer function $M(|\mathbf{V}_m|^2)$ and the calculated $|\mathbf{V}_m|^2$

value from the $|\mathbf{V}_m|^2$ computation program, the mapping controller generates two outputs to the 64-sample-point buffer: a sampling index and an enable signal. Since it is difficult to derive an analytical solution of the $|\mathbf{V}_m|^2$ pdf that is used in $M(|\mathbf{V}_m|^2)$, a simulation program was written for finding an approximate $|\mathbf{V}_m|^2$ pdf. Figure 4-18 shows the simulated result of the pdf, obtained from averaging 30 simulation outputs with 1.25×10^5 symbols/trial.

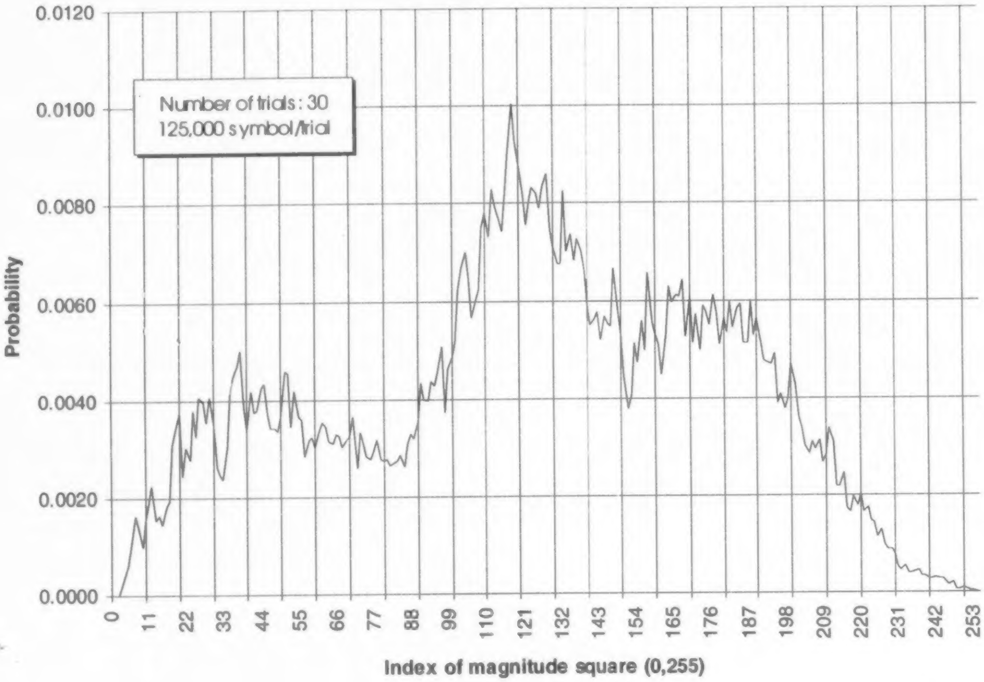


Figure 4-18. Simulated pdf obtained from 30 simulation outputs with 1.25×10^5 symbols/trial.

$M(|\mathbf{V}_m|^2)$ can now be obtained by dividing the simulated pdf into 64 equal area intervals; i.e., each interval has the same occurrence probability, and also has its corresponding sampling index. The number of the intervals is equal to the length of the 64-sampling-point buffer. In the implementation, a 256-point LUT, namely a mapping LUT indexed by $|\mathbf{V}_m|^2$, is used to map $M(|\mathbf{V}_m|^2)$. The content of the mapping LUT is the corresponding value of the sampling index. The memory size of this LUT is 256 words. Besides generating the sampling index, the mapping controller generates the enable signal, which is asserted once for every sampling index within a cycle of filling up the 64-sampling-point buffer. Another 64-point LUT, namely validation LUT indexed by the sampling index, is used to generate the enable signal. Two logical values are stored in this LUT: valid and invalid. The initial conditions of the validation LUT are all set to be valid; i.e., every sampling index can assert an enable signal. After an enable signal is asserted, the corresponding value in the validation LUT is assigned to be invalid. In this case, no enable signal will be generated for the same input sampling index until the validation LUT is reset by a reset signal generated from the 64-sampling-point buffer. Since the validation LUT represents only two possible conditions, its memory size is just 64×1 bit; i.e. 2 words. Figure 4-19 shows an implemented structure of the mapping controller.

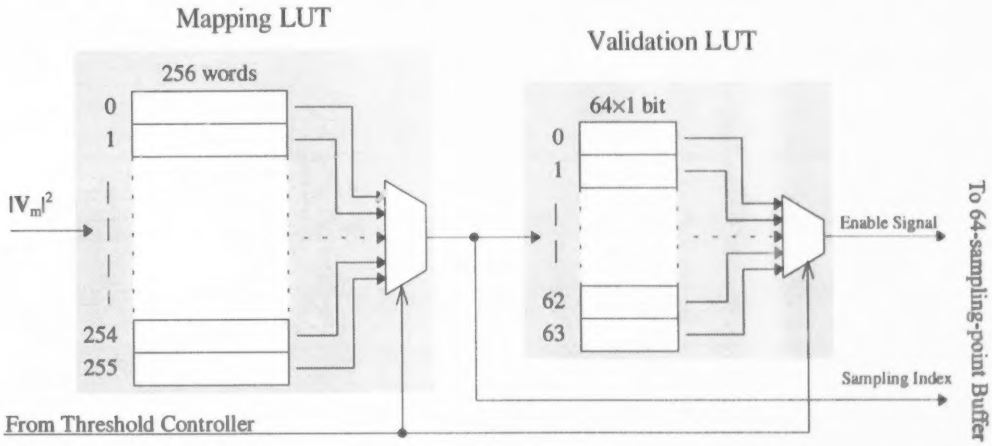

Mapping LUT
256 words
0
1
254
255
$|V_m|^2$
Validation LUT
64×1 bit
0
1
62
63
Enable Signal
Sampling Index
From Threshold Controller
To 64-sampling-point Buffer

Figure 4-19. Implemented structure of the mapping controller.

The 64-sampling-point buffer is the second buffer in the off-line distortion estimation unit. It is controlled by the mapping controller to latch 64 sets of the adjusted

$|\mathbf{V}_m|^2$, $\mathbf{V}_p$, and $\mathbf{V}_f$ data. These 64 sets of data cover the entire data range of $|\mathbf{V}_m|^2$, and are passed to the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator to calculate a new set of predistortion values. The memory size of this buffer is 5×64 word, where 5 is the number of words to represent one real and two complex data. This size of memory is allocated from the circular buffer, which will be described later. When the 64-sampling-point buffer has been filled, the entire content of this buffer is returned to the circular buffer. A new empty memory block for the 64-sampling-point buffer is then allocated from the circular buffer again. At the same time, a reset signal is also sent to the mapping controller to reset the validation LUT. In this case, the sampling process can continuously fill the sampled data into the 64-sampling-point buffer.

The circular buffer is used to interface between the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator and the 64-sampling-point buffer. The circular buffer contains 10 equal memory segments. Each memory segment provides a storage to the 64-sampling-point buffer in order to save the sampled data continuously. The size of each memory segment is the same as that of the 64-sampling-point buffer. The 10 memory segments inside the circular buffer are linked together to form a circular uni-directional access path. Figure 4-20 shows the structure of this buffer.

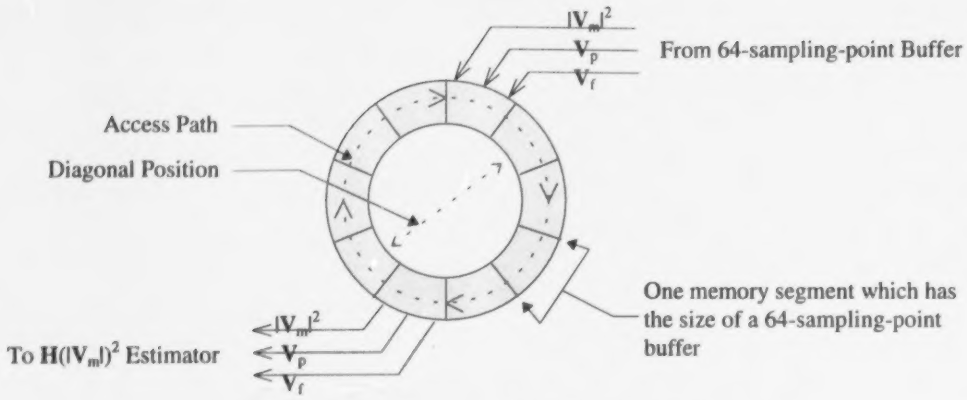

$|\mathbf{V}_m|^2$
$\mathbf{V}_p$
$\mathbf{V}_f$
From 64-sampling-point Buffer
Access Path
Diagonal Position
One memory segment which has the size of a 64-sampling-point buffer
$|\mathbf{V}_m|^2$
To $\mathbf{H}(|\mathbf{V}_m|)^2$ Estimator
$\mathbf{V}_p$
$\mathbf{V}_f$

Figure 4-20. Structure of the circular buffer.

Since the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator and the 64-sampling-point buffer are operated asynchronously, it is possible that both of them use one memory segment at the same time. In this situation, a conflict of updating the new sampled data and accessing the old sampled data at the same memory segment occurs. The following implementation procedure for the circular buffer is used to solve this problem.

- *No conflict occurred*
  - *Memory segment is allocated to the 64-sampling-point buffer sequentially.*
  - *Memory segment is accessed by the* $\mathbf{H}(|\mathbf{V}_m|^2)$ *estimator sequentially.*
- *Conflict occurred*
  - *If the 64-sampling-point buffer works faster than the* $\mathbf{H}(|\mathbf{V}_m|^2)$ *estimator, next memory segment provided to the 64-sampling-point buffer is allocated at the diagonal position of the memory segment which the* $\mathbf{H}(|\mathbf{V}_m|^2)$ *estimator is using.*
  - *If the* $\mathbf{H}(|\mathbf{V}_m|^2)$ *estimator works faster than the 64-sampling-point buffer, the* $\mathbf{H}(|\mathbf{V}_m|^2)$ *estimator has to wait for the 64-sampling-point buffer to finish the update.*

With this procedure, the circular buffer can continuously allocate a memory segment to the 64-sampling-point buffer, and does not disturb the operation of the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator. The memory size for implementing the circular buffer is $10\times((5\times64)+2)=3.22$ kwords, where +2 is the implementation overhead for the structure of the circular buffer.

The remainder of the four software programs in the off-line predistortion estimation unit are the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, the linear interpolator, the weighted averaging filter, and the $|\mathbf{V}_m|^2$ generator. The $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator, the linear interpolator and the weighted averaging filter are the core of this processing unit. They are used for generating a new predistortion value according to the sampled $\mathbf{V}_m$, $\mathbf{V}_p$, and $\mathbf{V}_f$ signals. Their characteristics have been completely described in Sections 3.2 and 3.4. Fortunately, they are simple mathematical equations, shown in Eqs. (3-10), (3-11) and (3-12), which can be applied directly into the software implementation. During the implementation, the coefficients of '*a*' and '*b*' in Eq. (3-12) are set to be 15/16 and 1/16, respectively. In order to coordinate with the clipping control unit, the power regulation unit and the loop selector, the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator has a minor modification in its implementation. Instead of using a fixed value of gain constant (K), a gain factor (K') is used. The value of K' is varied, based on the current output power level or the output signal level. As a result, the output power or signal level can be adaptively adjusted to a specific level.

The $|\mathbf{V}_m|^2$ generator is used to generate an incremental sequence which covers the entire value range of $|\mathbf{V}_m|^2$. This sequence causes the above three programs, the predistortion LUT buffer, and the clipping control unit to produce a new predistortion LUT. The value of the generator is increased by one step constant when the weighted averaging filter produces a new predistortion value. The step constant is obtained from the following equation:

$$\text{Step Constant} = 2^{\left(\left\lceil \log_2\left(|\mathbf{V}_m|^2{}_{\max}\right)\right\rceil - \left\lceil \log_2\left(\ \mathrm{TS}\ \right)\right\rceil\right)} \tag{4-12}$$

where $\lceil\ \rceil$ is a ceiling operator which returns the smallest integer greater than or equal to its argument, $|\mathbf{V}_m|^2{}_{\max}$ is the maximum value of $|\mathbf{V}_m|^2$ generated by the $\pi/4$ DQPSK encoder and pulse shaping filter, and TS is the size of the predistortion LUT. Since the value of $|\mathbf{V}_m|^2{}_{\max}$ was found to be 536,384,378, and the TS is set to be 1024, the step constant is calculated to be 524,288 by substituting the value of TS and $|\mathbf{V}_m|^2{}_{\max}$ into Eq. (4-12). The $|\mathbf{V}_m|^2$ generator is reset to a starting value when it is initialized or the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator has completed the calculation of a new block of sampled data. The starting value is set to be half of a step constant, so that the generated sequence is aligned to the center in each interval of the $|\mathbf{V}_m|^2$.

The predistortion LUT buffer is the last component in the off-line predistortion estimation unit. It consists of two predistortion LUTs. One LUT is shared by the real-time predistortion unit and the LUT buffer, and provides predistortion values to the vector multiplier and the weighted averaging filter. Since the vector multiplier and the weighted averaging filter have their own access path to this LUT, there is no conflict in sharing one predistortion LUT by two processing units. Another predistortion LUT in this buffer can only be accessed by this processing unit, and is used to store the most updated predistortion values, generated by the weighted averaging filter. When this LUT has been updated, the position of the two predistortion LUTs is swapped. As a result, a more efficient updating process for the predistortion LUT can be achieved. The initial value of both predistortion LUTs is set to 1+j·0; i.e., no predistortion signal is generated from the real-time predistortion unit before the first set of predistortion values is calculated.

4.3.3 Power Regulation Unit

The power regulation unit consists of three software programs, namely the power computation program, power combiner, and power control program. These programs cooperate with the off-line distortion estimation unit to keep the power of the output signal at a specific level. The power computation program continuously receives the sampled $\mathbf{V}_f$ signal from the output of the DACs in every sampling cycle, and applies Eq. (4-13) to compute an approximated value of the averaged power of I_f and Q_f ($\tilde{P}_{I_f}$ and $\tilde{P}_{Q_f}$) within one predistortion LUT update cycle (T_{LUT}).

$$\begin{aligned} \tilde{P}_{I_f} &= \frac{1}{N} \cdot \sum_{1}^{N} \left(I_f\right)^2 , \\ \tilde{P}_{Q_f} &= \frac{1}{N} \cdot \sum_{1}^{N} \left(Q_f\right)^2 \end{aligned} \tag{4-13}$$

where N is the number of sampling cycles within one T_{LUT}, and the resistance of the load is normalized to 1. Since the T_{LUT} is a constant, the value of N is also fixed. Therefore, the value of $1/N$ can be pre-determined, and stored into the DSP memory. As a result, the division for calculating the value of $1/N$ can be eliminated. Because of the simplicity of Eq. (4-13), the power computation program can directly use this equation for the implementation. The values of $\tilde{P}_{I_f}$ and $\tilde{P}_{Q_f}$ are output to the power combiner in every T_{LUT}. The power combiner is used to calculate the averaged output power of the $\mathbf{V}_f$, i.e.,

$\tilde{P}_{V_f}$, in every T_{LUT}. Since the $\tilde{P}_{V_f}$ is equal to the sum of $\tilde{P}_{I_f}$ and $\tilde{P}_{Q_f}$, the power combiner is just an adder to sum the received $\tilde{P}_{I_f}$ and $\tilde{P}_{Q_f}$. The power combiner then passes the value of $\tilde{P}_{V_f}$ to the power control program, which is implemented using an exponential integrator [42], as shown in Eq. (4-14):

$$\begin{aligned} e_p &= \tilde{P}v_f - P_{ref} \\ K'_i &= K'_{i-1} \cdot \left(1 - \xi \cdot e_p\right) \end{aligned} \tag{4-14}$$

where *i* is the iterative index with a period of T_{LUT}, *K'* is the gain factor, ξ is the damping ratio of the exponential integrator, e_p is the error between $\tilde{P}v_f$ and P_{ref}, and P_{ref} is the specific power reference level. According to the input $\tilde{P}v_f$, the power control program calculates a new value of *K'*. This new *K'* is then sent to the $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator through the loop selector. The $\mathbf{H}(|\mathbf{V}_m|^2)$ estimator uses the new *K'* to adjust the gain in the newly generated predistortion LUT, resulting in the transmitter signal power being limited to P_{ref}. Since $\tilde{P}v_f$ is obtained in every T_{LUT}, Eq. (4-14) is computed once in every T_{LUT}. The iteration will stop when the value of e_p with respect to P_{ref} is less than 0.2%. Also, because of its simplicity, Eq. (4-14) can be applied directly to the implementation of the power control program. In the implementation, the value of ξ is set to be 1/16, and the value of P_{ref} is set to be 50% above the averaged output power from the π/4 DQPSK encoder and pulse shaping filter.

4.3.4 Clipping Control Unit

The clipping control unit is similar to the power control unit. It consists of three software programs: a magnitude squared computation program, a maximum searching program, and a clipping control program. These programs are used to limit the output signal to its maximum level. Because the DSP does not have a square root function, it is better to use the magnitude squared of the signal for the computation throughout this processing unit, rather than using the amplitude of the signal. The magnitude squared computation program receives two values: the value of $|\mathbf{V}_m|^2$ from the $|\mathbf{V}_m|^2$ generator and the corresponding predistortion value from the weighted averaging filter. The output from this program is obtained by multiplying the input $|\mathbf{V}_m|^2$ to the magnitude squared of the corresponding predistortion value; i.e., the output value is equal to $|\mathbf{V}_p|^2$ at the input value of $|\mathbf{V}_m|^2$. Since the $|\mathbf{V}_m|^2$ generator continuously provides an incremental sequence of $|\mathbf{V}_m|^2$ within one T_{LUT}, the maximum searching program, in this case, is used to check through every output value from the magnitude squared computation program in order to find the maximum $|\mathbf{V}_p|^2$, ($|\mathbf{V}_p|^2_{\max}$). The clipping control program then substitutes the maximum value of $|\mathbf{V}_p|^2_{\max}$ in an exponential integrator to calculate a new K' in every T_{LUT}:

$$\begin{gathered} e = \left|V_p\right|^2_{\max} - \left(\rho \cdot 2^{15}\right)^2 \\ K'_i = K'_{i-1} \cdot \left(1 - \xi \cdot e\right) \end{gathered} \tag{4-15}$$

where i is the iterative index with a period of T_{LUT}, 2^{15} is the absolute maximum output value in the 16-bit D/A converters, ρ is the output tolerance of the DACs, e is the error of between $|\mathbf{V}_p|^2_{\max}$ and the maximum output reference, and ξ is the damping ratio of the loop. In the power control program, the clipping control program calculates a new value of *K'* in order to adjust the gain in the new predistortion LUT. As a result, the output signal level is limited to the maximum DAC output. Similarly, Eq. (4-15) also iterates once for every T_{LUT}, and is implemented directly in the clipping control program. In the implementation, the value of ξ is set to be 1/16, and the value of ρ is set to be 90%. The iteration of Eq. (4-15) will stop when the value of *e* with respect to the $(\rho \cdot 2^{15})^2$ is less than 0.2%.

4.3.5 Loop Selector

The loop selector is used to determine which one of the two control loops (i.e., the power regulation loop and the clipping control loop) is activated. It consists of two components: a loop switching program and a gain factor buffer. The loop switching program receives three input data in every T_{LUT}: the $|\mathbf{V}_p|^2_{max}$ from the clipping control unit, and two gain factors from the power regulation unit and clipping control unit. This program counts the number of times the input value of the $|\mathbf{V}_p|^2_{max}$ exceeds the absolute maximum output value in the 16-bit D/A converters, i.e. 2^{15}. If the total number of exceeded values is larger than a given reference number, the loop switching program stops the current control loop, and lets another control loop to take over. The corresponding gain factor is then passed to the gain factor buffer. In the beginning, the power regulation control loop is first activated in order to keep the averaged output power at a specific level. If the power regulation control loop should fail for any reason, then the clipping control loop takes over and limits $\mathbf{V}_p$ below the maximum input value of the DACs. Since the operation between the main linearization loop and the selected control loop is asynchronous, the loop switching program cannot directly pass the gain factor to the main linearization loop. The gain factor buffer is used to interface between the loops, and to provide a non-blocking transfer for the gain factor.

4.4 Software Architecture of the Enhanced Linearizer

As shown in Figures 4-12 and 4-13, the entire linearizer is realized by five processing units: the real-time predistortion unit, the off-line distortion estimation unit, the power regulation unit, the clipping control unit, and the loop selector. Each processing unit consists of several programs and data buffers. Based on their execution characteristics, the programs, implemented in each processing unit, can be classified into two categories: real-time program and off-line program. Either the real-time or the off-line programs may control the other programs within one processing unit or across the boundary of the processing units. In addition, some of the data buffers are implemented to share their contents by the programs or processing units. In this case, it is difficult to apply sequential programming techniques to implement the programs and buffers in the corresponding processing units. Large computation power is required to execute every implemented program sequentially. Moreover, a complicated flow control method in each program and data buffer is also required.

In order to reduce the time for algorithm development, verification, and testing as well as to optimize the processing speed, OOP and nonpreemptive multitasking techniques have been adopted in the linearizer's implementation. As a result, the complicated processing units, shown in Figure 4-13, can be organized into five different program objects: real-time processing object, power computation object, sample processing object, clipping control object and linearization processing object. Figure 4-21 shows the software architecture of the enhanced linearizer. Table 4-3 shows the corresponding programs, which are implemented into the program objects.

Each program object performs a particular job. The real-time processing object performs the processes that must be executed in real-time, such as predistorting the input quadrature modulated signal, latching the sampled $\mathbf{V}_m$, $\mathbf{V}_P$, and $\mathbf{V}_f$ signals, pushing the latched data into the delay adjusting buffer, and calculating the accumulated energy of the signal. The power regulation object and the clipping control object are used to realize the power regulation unit and the clipping control unit, respectively. The sampling processing object is used to process the sampled signals before they are passed to the linearization objects. The linearization object performs the linearization algorithm, and generates a new predistortion LUT based on the new sampled data. Besides these five program objects, an additional program object, namely initialization object, is used to initialize the working environment for the entire linearizer. This object only executes once, and is replaced by the predistortion LUT after finishing initialization. Therefore, the size of the linearizer's program can be reduced at run-time.

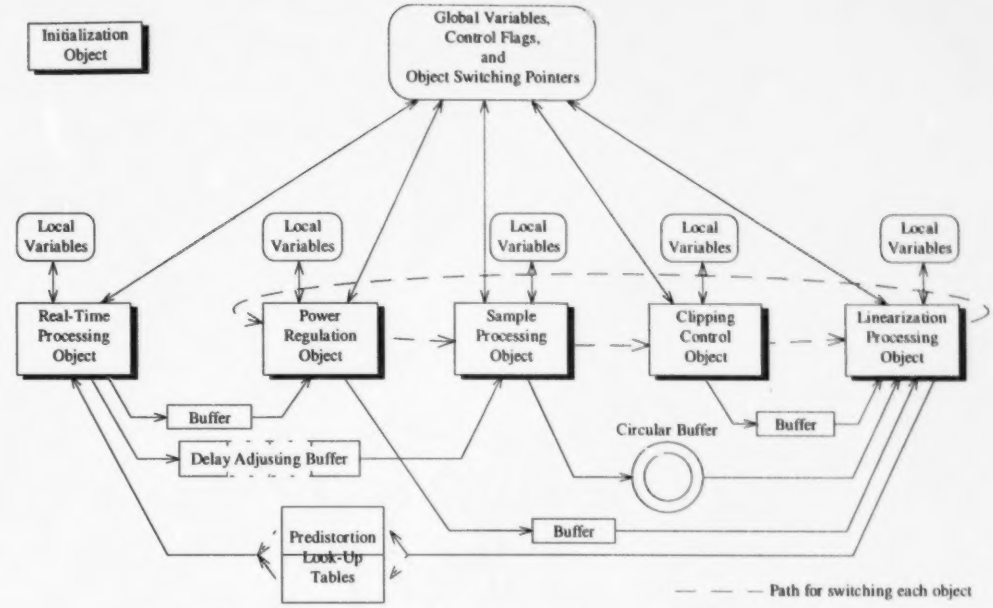

Initialization Object
Global Variables, Control Flags, and Object Switching Pointers
Local Variables
Local Variables
Local Variables
Local Variables
Local Variables
Real-Time Processing Object
Power Regulation Object
Sample Processing Object
Clipping Control Object
Linearization Processing Object
Buffer
Delay Adjusting Buffer
Circular Buffer
Buffer
Predistortion Look-Up Tables
Buffer
Path for switching each object

Figure 4-21. Software architecture of the enhanced linearizer.

Table 4-3. ARRANGEMENT BETWEEN PROGRAM OBJECTS AND PROGRAMS.

Program Object	Involved Program
Real-Time Processing Object	$|\mathbf{V}_m|^2$ Computation Program, Predistortion LUT, Vector Multiplier, Delay Adjusting Buffer Control Program, and Part of Output Power Computation Program.
Power Regulation Object	Part of Output Power Computation Program, Power Control Program, and Power Compensator.
Sample Processing Object	Discrete Time Delay Compensator, Fractional Time Delay Compensator, Threshold Controller, Mapping Controller, 64-sampling-point Buffer Controller Program, and Circular Buffer Controller Program.
Clipping Control Object	Magnitude Squared Computation Program, Maximum Searching Program, and Clipping Control Program.
Linearization Processing Object	$\mathbf{H}(|\mathbf{V}_m|^2)$ Estimator, Linear Interpolator, Weighted Averaging Filter, Predistortion LUT, $|\mathbf{V}_m|^2$ Generator, Gain Factor Buffer, and Loop Switch Program.

Except for the real-time processing object, the other four program objects are designed to operate in the nonpreemptive multitasking structure. In this case, a program, implemented in a program object, is divided into numbers of small tasks. In addition, each program object has its own switching pointer, which points to a task in another object. When one task in an object is completed, the execution of this object is interrupted by itself, and is switched to the task that the switching pointer points to. As a result, the computational resource can be shared by each program object. Unlike using a sequential programming technique, the processor needs not going through all programs in the linearizer in order to produce one output; it only executes those tasks which are required in the current situation. Moreover, the switching pointer provides a flexible flow control on the task execution. Figure 4-22 shows a sample execution trace of the implemented linearizer program.

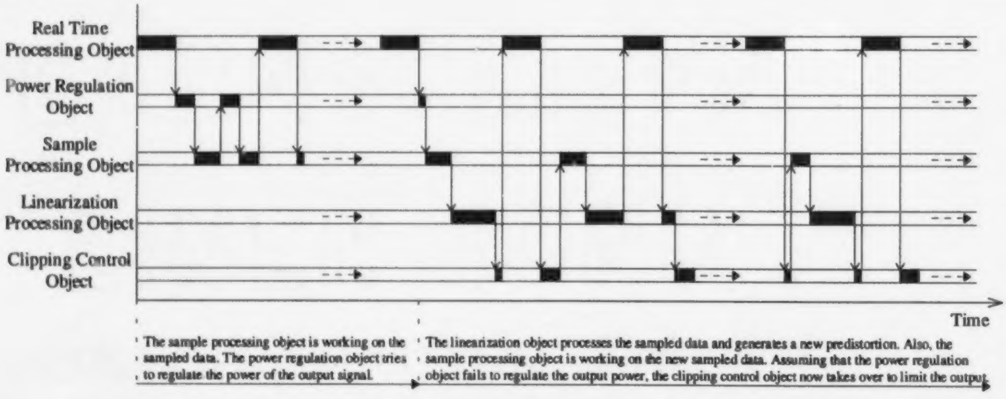

Real Time Processing Object
Power Regulation Object
Sample Processing Object
Linearization Processing Object
Clipping Control Object
Time
The sample processing object is working on the sampled data. The power regulation object tries to regulate the power of the output signal.
The linearization object processes the sampled data and generates a new predistortion. Also, the sample processing object is working on the new sampled data. Assuming that the power regulation object fails to regulate the output power, the clipping control object now takes over to limit the output.

Figure 4-22. Sample execution trace of the implemented linearizer program.

The nonpreemptive multitasking system cannot be used for implementing the real-time processing object. The execution of the programs inside this object must be completed within one sampling cycle; i.e., the execution of this object cannot be shared by the others. Therefore, the size of the real-time processing object should be designed as small as possible in order to free more computational resource for the other program objects. In this case, all of the required functions in the real-time processing unit (see Table 4-3) are integrated into one program, resulting in an optimized 63 instruction real-time program. This program is small enough to be completely pushed into the DSP's cache memory. As a result, a faster processing speed for the real-program can be achieved.

In the original plan, the entire enhanced linearizer was to be implemented into one TMS320C30 DSP, and the linearization program was to be written by DSP Assembly language [40, 43, 44]. However, because of the time limitations, part of the linearization object, the power regulation object, and the clipping control object were transferred to an IBM compatible 486 computer. The transferred program objects were written in C++ programming language [38, 39, 45, 46, 47, 48]. The remaining program objects and the transferred program objects are linked together by three components: communication object, hardware interface, and hardware interfacing class. The communication object and the hardware interfacing class were implemented using a DSP assembly program and a C++ program, respectively. The hardware interface is realized by digital circuits. Since all of the digital circuits have already been built in the TMS320C30 DSP development platform, no special hardware design is necessary. These three components provide a bi-directional communication channel between the split program objects. Figure 4-23 shows a structure diagram of the communication between the C30 DSP and the 486 PC.

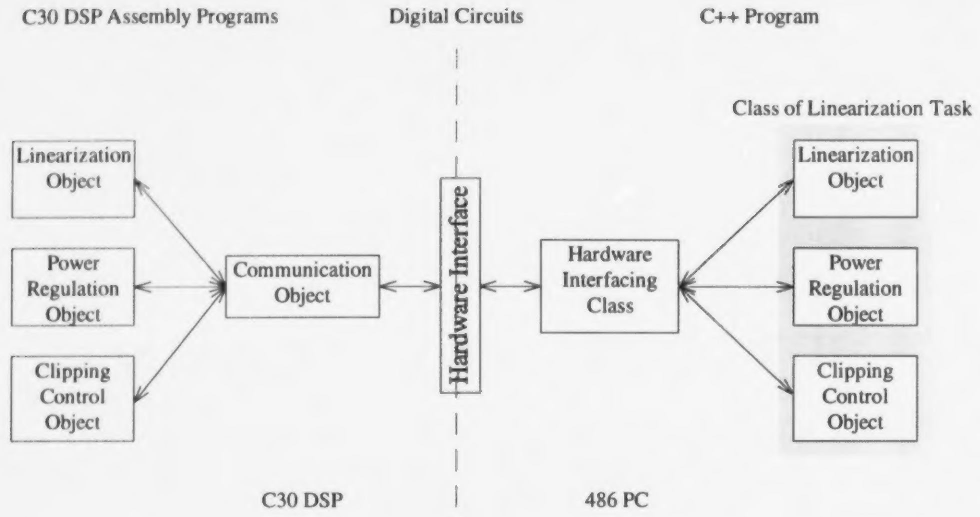

C30 DSP Assembly Programs
Digital Circuits
C++ Program
Class of Linearization Task
Linearization Object
Power Regulation Object
Clipping Control Object
Communication Object
Hardware Interface
Hardware Interfacing Class
Linearization Object
Power Regulation Object
Clipping Control Object
C30 DSP
486 PC

Figure 4-23. Structural diagram of the communication between the C30 DSP and the 486 PC. .

Besides implementing part of the program objects in the 486 computer, the 486 computer was also used to download the DSP program into the TMS320C30 development platform, and to measure the performance of the linearizer. Figure 4-24 shows a hierarchical diagram of the implemented C++ programs. The hardware interfacing class is used to interface among the DSP hardware, the 486 computer and the C++ programs. The class of linearization task is the C++ program objects that are equivalent to the transferred DSP program objects. The class of implemented linearizer is used to download the DSP program into the development platform, to initialize the C++ program, and to start the entire linearizer. The rest of the class objects are used for development and verification. The memory map and programming list of the DSP programs and the C++ programs are shown in Appendix D, and Appendix E List 2 - 7 and 9 - 18, respectively.

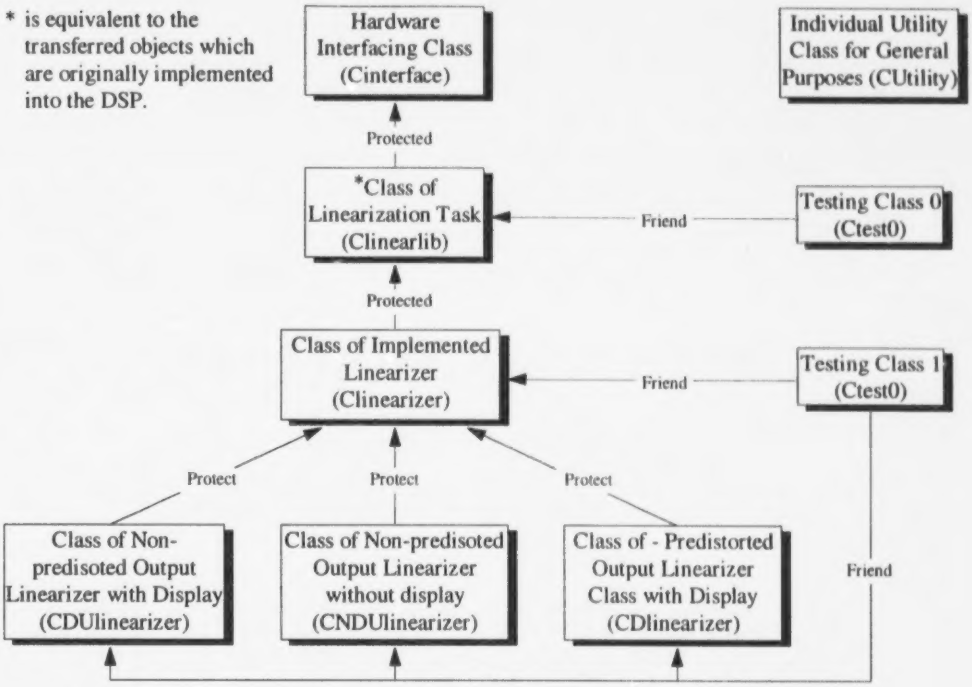

* is equivalent to the transferred objects which are originally implemented into the DSP.
Hardware Interfacing Class (Cinterface)
Individual Utility Class for General Purposes (CUtility)
Protected
*Class of Linearization Task (Clinearlib)
Friend
Testing Class 0 (Ctest0)
Protected
Class of Implemented Linearizer (Clinearizer)
Friend
Testing Class 1 (Ctest0)
Protect
Protect
Protect
Class of Non-predisoted Output Linearizer with Display (CDUlinearizer)
Class of Non-predisoted Output Linearizer without display (CNDUlinearizer)
Class of - Predistorted Output Linearizer Class with Display (CDlinearizer)
Friend

Figure 4-24. Hierarchical diagram of the implemented C++ program objects.

Although implementing part of DSP program object in a 486 computer can significantly reduce the time for development and verification, the trade-off is the time efficiency of the enhanced linearizer. The slow hardware and software interface between the C30 DSP and 486 computer, i.e., the communication object, the hardware interfacing class, and the hardware interface, reduce the processing efficiency.

4.5 Quadrature Modulator, Power Amplifier and Quadrature Demodulator

In the original plan, the quadrature modulator, power amplifier and quadrature demodulator were to be completely implemented using electronic devices, and used to convert the baseband quadrature modulated signal to a transmitting RF signal, to amplify the RF signal and send the signal to a load, as well as to convert the amplified RF signal back to baseband. However, because of the time limitations of this project, the circuit implementation of this part has been realized by two base-band testing loads in order to simulate the characteristics of a RF power amplifier: a base-band digitally emulated power amplifier and a base-band op-amp diode circuit. Since these testing loads are operated at base-band, no quadrature modulator and demodulator is required in this implementation.

The base-band digitally emulated power amplifier is completely implemented inside the DSP. It consists of a magnitude squared computation program, a vector multiplier and an amplifier characteristic LUT. The block diagram of the digitally emulated power amplifier is shown in Figure 4-25.

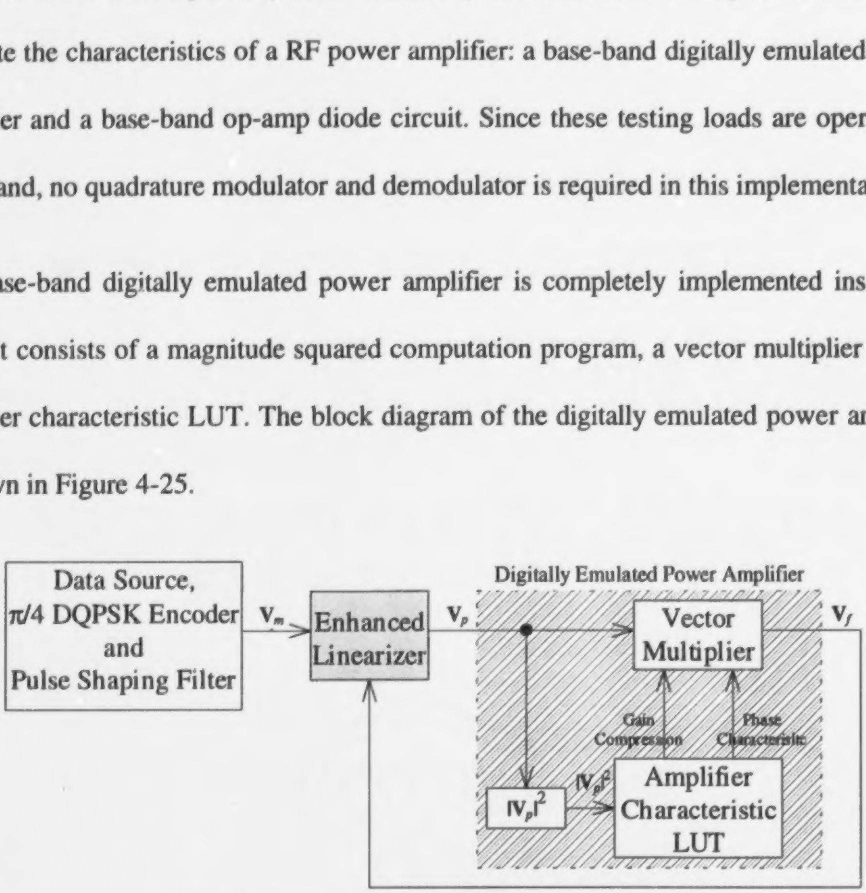

Data Source,
π/4 DQPSK Encoder
and
Pulse Shaping Filter
V_m
Enhanced Linearizer
V_p
Digitally Emulated Power Amplifier
Vector Multiplier
V_f
Gain Compression
Phase Characteristic
$|V_p|^2$
$|V_p|^2$
Amplifier Characteristic LUT

Figure 4-25. Block diagram of the digitally emulated power amplifier.

The amplifier characteristic LUT is a one-to-two LUT, which is indexed by the magnitude squared of the complex input signal, and contains the values of the gain compression and phase characteristics of a class B RF power amplifier. These values are obtained by converting the transfer characteristic of an actual class B RF power amplifier. Figure 4-26 and Appendix C, Table C-1 show the measured transfer characteristic of the amplifier [49]. Since the low input voltage measurement is missing from the measured transfer characteristic, and also since the measured data do not provide sufficient resolution to tabulate the LUT, a best curve fit of the measured data is used to generate a complete LUT. As a result, the amplifier characteristic LUT contains 4096 points to represent the characteristic of the power. Figure 4-27 shows the gain compression and the phase characteristic of the power amplifier, which are stored in the amplifier characteristic LUT.

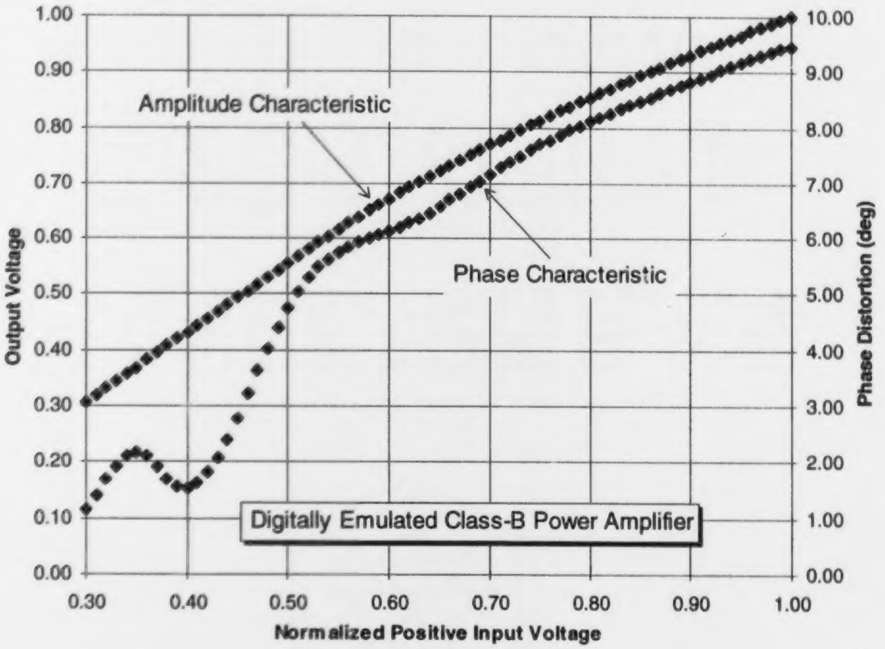


Figure 4-26. Complex transfer characteristic of the emulated class B power amplifier.

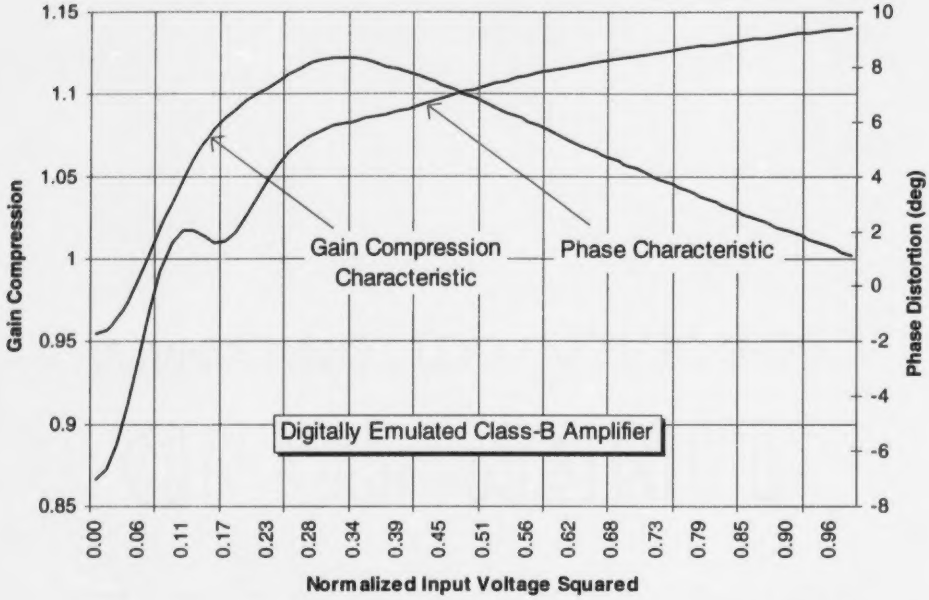


Figure 4-27. Gain compression and phase characteristics of the emulated class B power amplifier.

The $\mathbf{V}_p$ signal, generated by the linearizer, is connected to the magnitude squared computation program and the vector multiplier. The result from the magnitude squared computation program then indexes to the amplifier characteristic LUT, and locates a pair of values, which correspond to the gain compression and phase characteristic at this input signal level. The vector multiplier generates the amplified quadrature signal $\mathbf{V}_f$ by multiplying the $\mathbf{V}_p$ by the corresponding LUT value. The DSP program for realizing this emulated power amplifier is shown in Appendix E List 19.

The base-band op-amp diode circuit consists of two low noise operational amplifiers and a diode network. The circuit diagram is shown in Appendix C, Figure C-1. This circuit is directly connected to the ADC and DAC, and only simulates the AM-AM distortion in a RF power amplifier. Because the diode op-amp circuit cannot distort both the I and Q channels by the proper amount at the same time, only one of the I or Q signals from the linearizer's output is connected to this circuit; the signal in the other channel, in this case, is set to be zero. The amplitude transfer characteristic of this circuit is shown below.

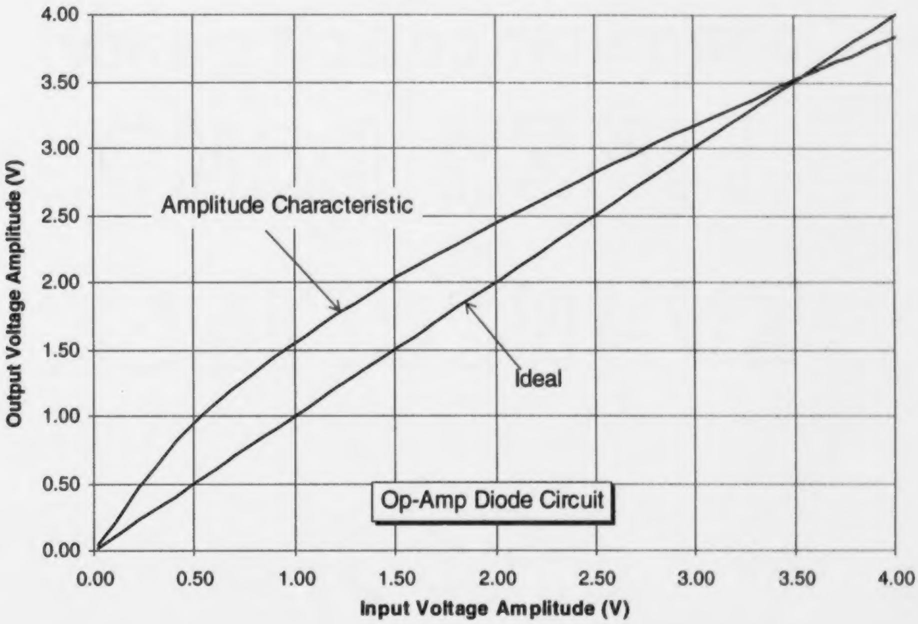


Figure 4-28. Amplitude transfer characteristic of the op-amp diode circuit.

5. RESULTS

This chapter gives the test results for the performance of the enhanced linearizer. The sampling rate and the related parameters are first described in Section 5.1. Then, evaluation of the RF power amplifier linearity is presented. The last two sections give the results for the spectral tests for the π/4 DQPSK encoder and pulse shaping filter, and the enhance linearizer using two simulated forms of the power amplifier.

5.1 Sampling Rate and Related Parameters

The sampling rate is the key parameter in the digitally implemented linearizer. It directly affects the input data bit rate, the bandwidth of the transmitted signal, and the overall performance of the transmitter. The maximum sampling rate is limited by the conversion rate of the ADC/DAC, the performance of the π/4 DQPSK encoder and pulse shaping filter, and the performance of the enhanced linearizer. The maximum conversion rate of the ADC/DAC, provided by the DSP development platform, is 200 kHz [40]. The π/4 DQPSK encoder and pulse shaping filter were designed to be operated at the rate of ~200 ksample/s (see Section 4.2). The performance of the enhanced linearizer is dependent on three factors: the processing speed of the TMS320C30 DSP, the software architecture of the linearizer, and the optimal level of the implemented DSP programs. Except for the processing speed of the DSP, there is no simple relationship to characterize the actual performance of the other two factors. In this case, if the enhanced linearizer is assumed to be fast enough to process every sample, then the maximum sampling rate is obviously equal to 200 ksample/s. However, the actual maximum sampling rate, obtained from the experiments, is equal to ~151.2 ksample/s. In addition to the sampling rate of the transmitter, the oversampling rate of pulse shaping filter (*M*), described in Section 4.2.2, is also important. According to Section 4.2.2, the value of *M* should be greater than or equal to 10. Two values of *M* are chosen for testing the performance of the transmitter: 10 and 16. The following tables shows the related parameters for the selected value of *M* at the sampling rate of 151.2 ksample/s.

Table 5-1. VALUE OF THE INPUT BIT RATE AND TRANSMITTED SIGNAL BANDWIDTH WHEN M IS 10 OR 16, AND THE SAMPLING RATE IS 151.2 KSAMPLE/S.

Oversampling rate (*M*)	10 samples/symbol	16 samples/symbol
Input data bit rate	30.4 kb/s	18.9 kb/s
Transmitted signal bandwidth	17.0 kHz	10.6 kHz

The value of the input data bit rate and the transmitted signal bandwidth are calculated using (5-1) and (5-2), respectively.

$$\text{Input data bit rate} = 2\ \text{bit / symbol} \cdot \frac{151.2\ \text{ksample / s}}{M} \tag{5-1}$$

$$\text{Transmitted signal bandwidth} = \text{Input data bit rate} \cdot \frac{27.1\ \text{kHz}}{48.6\ \text{kb / s}} \tag{5-2}$$

where 27.1 kHz and 48.6 kb/s correspond to the bandwidth of the square-root-raised-cosine pulse shaping filter and the input data bit rate used in the IS-54 standard, respectively.

5.2 Evaluation for RF Power Amplifier Linearity

Measurement of spectral spreading of bandlimited input signals is one method of characterizing the linearity of an amplifier. The maximum acceptable level of power emission in the adjacent channels from the output of a RF power amplifier is specified by the IS-54 standard to be -26 dB, -45 dB, and -60 dB below the maximum power emission in adjacent channels 1, 2 and 3, respectively. To evaluate the linearity of a RF power amplifier, the out-of-band power emission of the amplifier must be compared with these specific power emission levels.

In the IS-54 standard, the channel spacing is defined to be 30 kHz; i.e., the center frequency of each channel is 30 kHz apart, for an input data bit rate of 48.6 kb/s. For the results reported here, tests were performed for 18.9 kb/s and 30.4 kb/s, as given in Table 5-1. To properly evaluate the results in terms of the interference in the adjacent channels, it is necessary to consider a channel spacing that is reduced in accordance with the bit rate. Eq.(5-3) shows the required relationship between the input data bit rate and the frequency separation. Table 5-2 shows the testing input data bit rate and the corresponding channel spacing.

$$\text{Channel Spacing} = (\text{Input data bit rate}) \cdot \frac{30\ \text{kHz}}{48.6\ \text{kb/s}} \tag{5-3}$$

Table 5-2. RELATIONSHIP BETWEEN THE TESTING INPUT DATA BIT RATE AND THE CORRESPONDING CHANNEL SPACING.

Input bit rate	18.9 kb/s	30.4 kb/s
Channel spacing	11.7 kHz	18.8 kHz

5.3 Test Results for the π/4 DQPSK Encoder and Pulse Shaping Filter

Two tests have been carried out in order to examine the performance of the π/4 DQPSK encoder and the pulse shaping filter. The setup for both tests is illustrated in Figure 5-1.

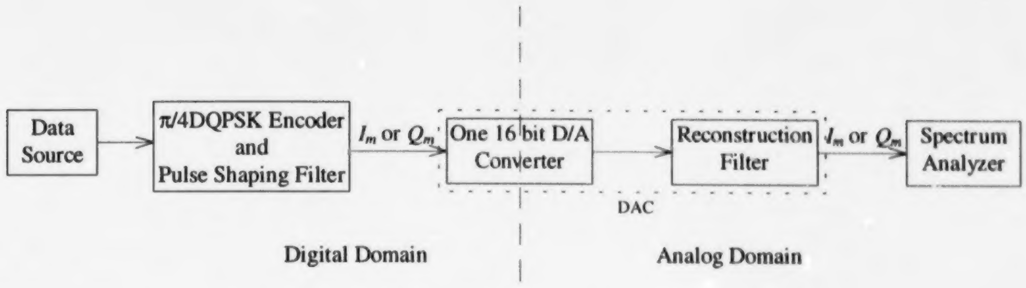

Data Source
π/4DQPSK Encoder and Pulse Shaping Filter
I_m or Q_m
One 16 bit D/A Converter
Reconstruction Filter
I_m or Q_m
Spectrum Analyzer
DAC
Digital Domain
Analog Domain

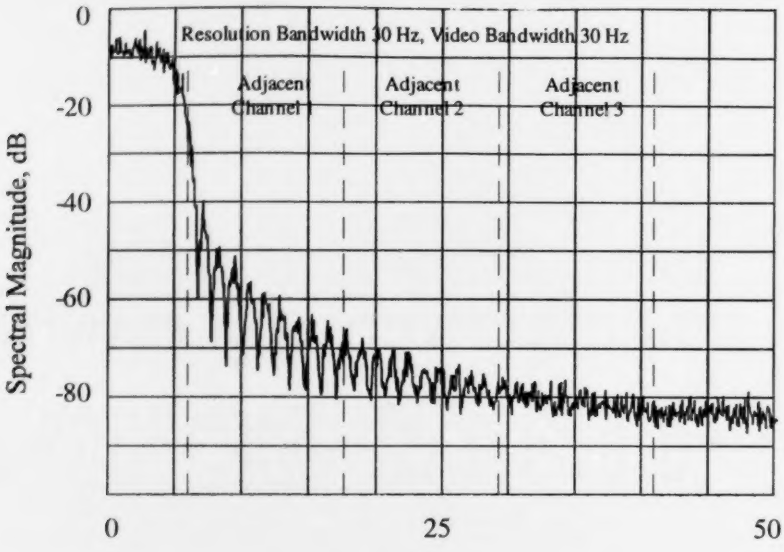
Figure 5-1. Test setup for the π/4 DQPSK encoder and pulse shaping filter.

Since the I and Q component of a quadrature signal $\mathbf{V}_m = I_m + j \cdot Q_m$ are orthogonal, I_m and Q_m should have the same spectral shape. Therefore, in these two tests, only one of the I or Q outputs signal from the π/4 DQPSK encoder and pulse shaping filter is connected to a 16 bit DAC. Then, a spectral analyzer is used to measure the signal spectrum from the output of the DAC. The specifications for these two tests is shown in Table 5-3. The results of the two tests are shown in Figures 5-2 and 5-3.

Table 5-3. SPECIFICATIONS OF THE TESTS FOR THE π/4 DQPSK ENCODER AND THE PULSE SHAPING FILTER.

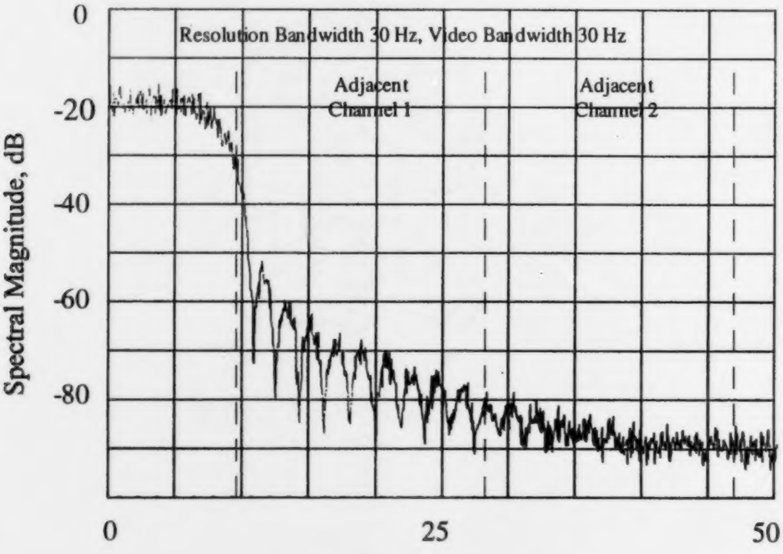
Test	Test Channel	Bit rate (kb/s)	Samples per symbol
1	I or Q channel	18.9	16
2	I or Q channel	30.4	10

As observed in both test results, the out-of-band power emission in adjacent channels 1, 2, and 3 is -50 dB, -60 dB, and -70 dB, respectively. These values are 24 dB, 20 dB, and 10 dB lower than the IS-54 standard maximum acceptable level of power emission in the corresponding adjacent channels. These results show that the performance of the π/4 DQPSK encoder and pulse shaping filter used to generate the $\mathbf{V}_m$ signal is excellent.



Frequency, kHz

Figure 5-2. Output spectrum of the π/4 DQPSK encoder and pulse shaping filter at a bit rate of 18.9 kb/s (Test 1).



Frequency, kHz

Figure 5-3. Output spectrum of the π/4 DQPSK encoder and pulse shaping filter at a bit rate of 30.4 kb/s (Test 2).

5.4 Test Results for the Enhanced Linearizer

Four sets of tests for the enhanced linearizer have been performed. These tests have been carried out using an op-amp diode circuit for single channel tests (i.e., only the I or Q signal, but not both), while digital emulation was used for two channel tests (i.e., both I and Q signals) to simulate the characteristic of the RF power amplifier. Figures 5-4 and 5-5 show the test setup for these simulated loads. Table 5-4 shows the specifications for each test. In these tests, the enhanced linearizer can achieve a LUT updating rate of 33 tables/s and a LUT convergence time of ~2 seconds. Moreover, the linearizer is able to regulate the output signal level to the required maximum.

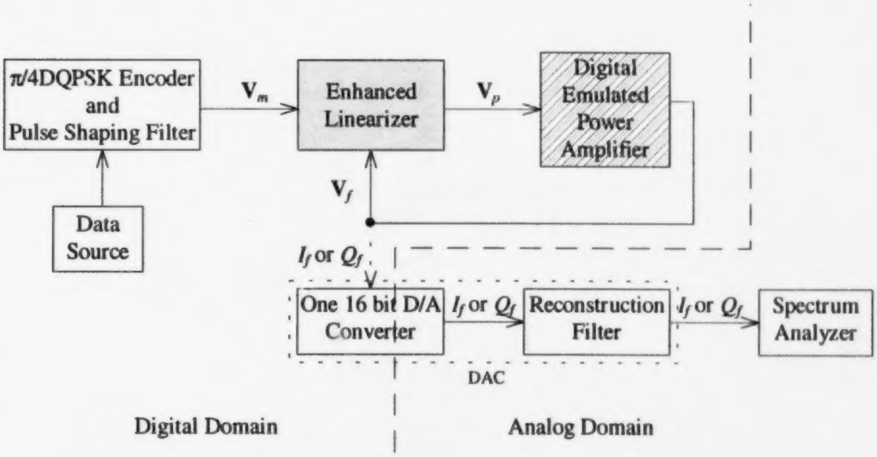

π/4DQPSK Encoder and Pulse Shaping Filter
V_m
Enhanced Linearizer
V_p
Digital Emulated Power Amplifier
V_f
Data Source
I_f or Q_f
One 16 bit D/A Converter
I_f or Q_f
Reconstruction Filter
I_f or Q_f
Spectrum Analyzer
DAC
Digital Domain
Analog Domain

Figure 5-4. Test setup for the digitally emulated power amplifier.

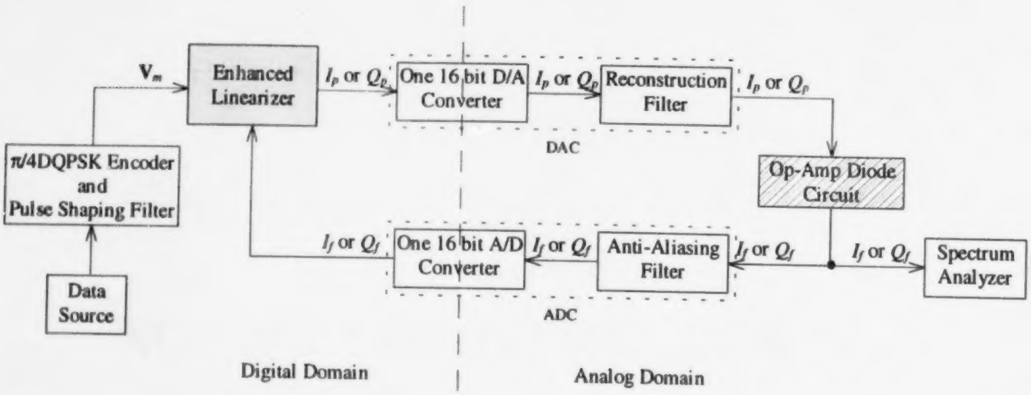

Vm
Enhanced Linearizer
I_p or Q_p
One 16 bit D/A Converter
I_p or Q_p
Reconstruction Filter
I_p or Q_p
DAC
π/4DQPSK Encoder and Pulse Shaping Filter
Op-Amp Diode Circuit
I_f or Q_f
One 16 bit A/D Converter
I_f or Q_f
Anti-Aliasing Filter
I_f or Q_f
I_f or Q_f
Spectrum Analyzer
Data Source
ADC
Digital Domain
Analog Domain

Figure 5-5. Test setup for the op-amp diode circuit.

Table 5-4. SPECIFICATIONS OF FOUR DIFFERENT TESTS FOR THE ENHANCED LINEARIZER.

Test	Testing circuit	Test Channel	Bit rate (kb/s)	Samples per symbol
1	Digitally emulated RF amplifier	I and Q channels	18.9	16
2	Op-amp diode circuit	I or Q channel	18.9	16
3	Digitally emulated RF amplifier	I and Q channels	30.4	10
4	Op-amp diode circuit	I or Q channel	30.4	10

Since the op-amp diode circuit cannot distort both the I and Q channels at the same time, only the I or Q signal is connected to this simulated load, not both; the signal in the other channel is, in this case, set to zero. Three outputs of power spectral density (psd) are recorded for each test: the output from the π/4 DQPSK encoder and pulse shaping filter, the output from the simulated load without enabling the linearizer, and the output from the simulated load when the linearizer is used. Figures 5-6, 5-7, 5-8 and 5-9 show the results of these four tests, respectively.

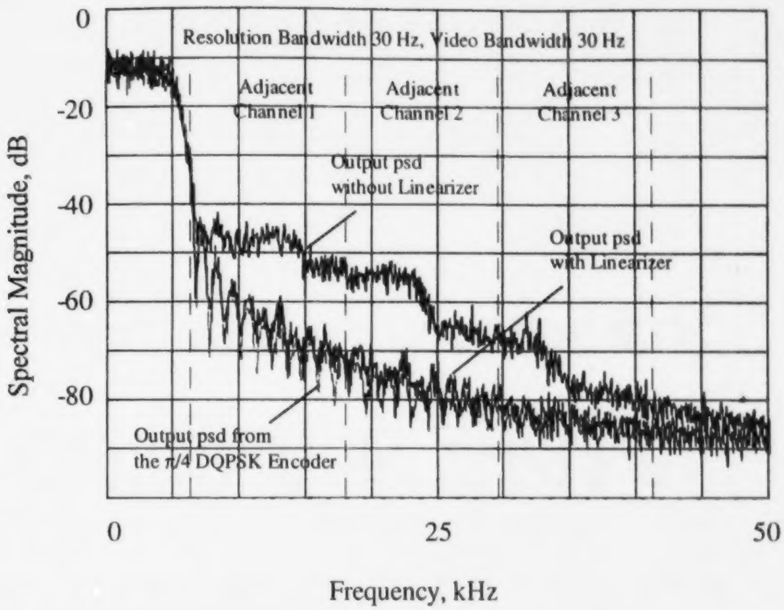


Figure 5-6. Output spectrum of the enhanced linearizer using the emulated power amplifier at a bit rate of 18.9 kb/s (Test 1).

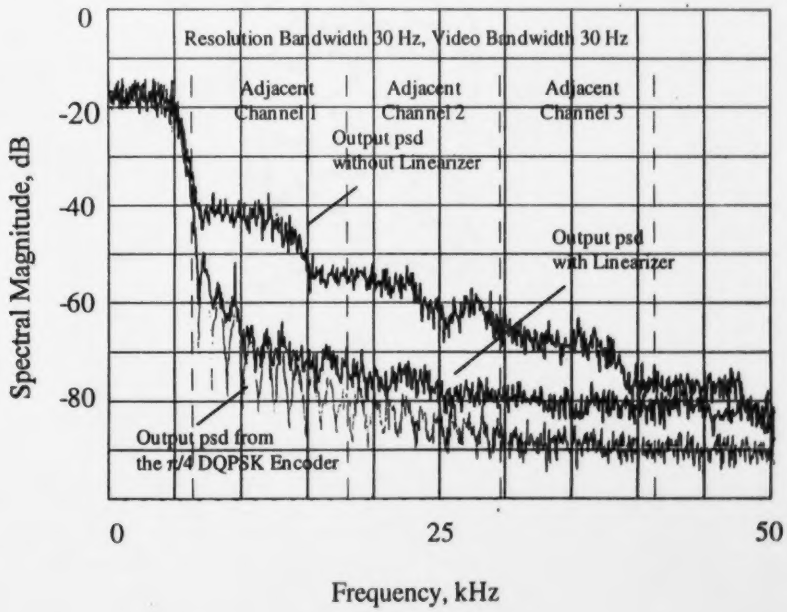


Figure 5-7. Output spectrum of the enhanced linearizer using the op-amp diode circuit at a bit rate of 18.9 kb/s (Test 2).

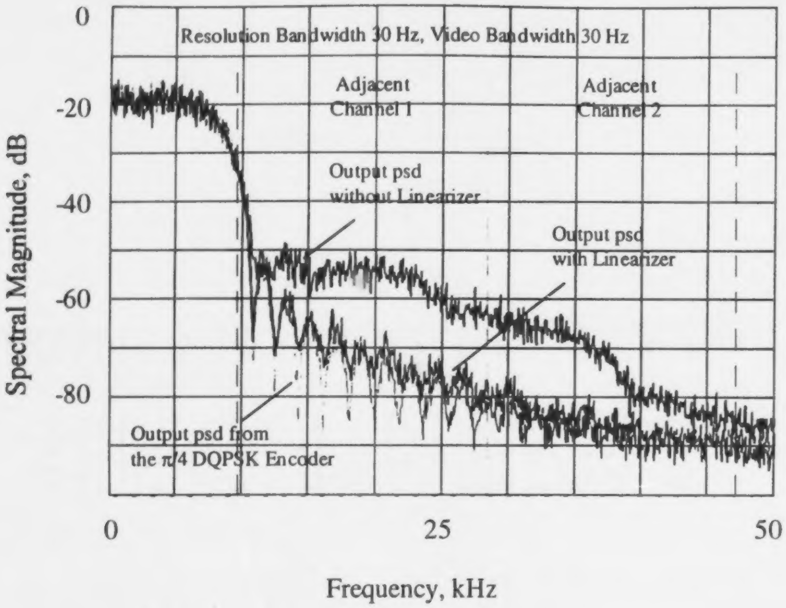


Figure 5-8. Output spectrum of the enhanced linearizer using the emulated power amplifier at a bit rate of 30.4 kb/s (Test 3).

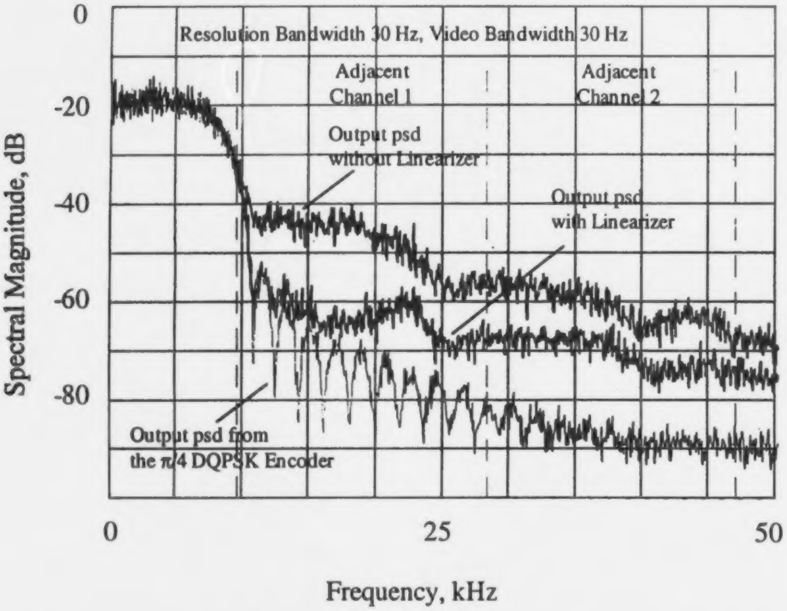


Figure 5-9. Output spectrum of the enhanced linearizer using the op-amp diode circuit at a bit rate of 30.4 kb/s (Test 4).

As observed in Figure 5-6 (i.e., Test 1), the enhanced linearizer works almost ideally when conversion error in the DAC, quantization noise in the ADC, non-ideal filter response in the anti-aliasing filter and reconstruction filter, and signal delay are not involved. The entire distorted spectrum generated by the digitally emulated power amplifier can be restored by the linearizer back to the original amplifier input psd. In Test 2, an op-amp diode circuit is used for generating the spectral distortion. The effects of noise, signal delay, non-ideal filter response, ADC and DAC, and imperfect magnitude dependence of the simulated load are included in this test. The enhanced linearizer still has good performance in restoring the distorted spectrum, especially in the first adjacent channel. Figure 5-7 shows that there is about a 28 dB reduction of the spectral distortion in the output signal in the first adjacent channel. However, the output spectrum in the second and higher adjacent channels is not restored completely to the amplifier input psd. Therefore, the effect of the magnitude and phase distortion of the anti-aliasing filter and of the reconstruction filter, as well as the frequency dependence of the op-amp diode circuit are observed to degrade the linearizer's performance.

Tests 3 and 4 are similar to the previous tests except that the bit rate and over-sampling rate have been changed to 30.4 kb/s and 10 symbol/s, respectively. As observed in Figure 5-8, the linearizer also almost completely restores the distorted spectrum for this higher bit rate, when the amplifier is emulated digitally. However, in Test 4, increasing the bit rate is observed to suppress the distorted output spectrum about 10 dB less than in Test 2, for frequencies above 15 kHz. Because the cut-off frequency of the anti-aliasing filter and of the reconstruction filter were not changed, this additional

10 dB distortion may be caused by the large phase variation of these filters. The linearizer cannot correct for this type of distortion, since the delay compensator can compensate only for the frequency independent delay.

6. SUMMARY AND CONCLUSIONS

A nonlinear RF power amplifier, such as a class B or class C power amplifier, has high power conversion efficiency. However, it also introduces two types of nonlinear distortion to the input signal, namely amplitude-to-amplitude (AM-AM) distortion and amplitude-to-phase (AM-PM) distortion. These distortions, caused by the nonconstant power gain and phase delay in the amplifier, result in spectral broadening of the output signal, especially for a linearly modulated signal such as a π/4 DQPSK signal. The spectral broadening of the signal, will interfere with signals in the adjacent RF channels. Therefore, in order to obtain a high power conversion efficiency without causing distortion of the signal, amplifier linearization is required.

Adaptive complex gain predistortion is a baseband linearization technique. Depending on the distortion caused by the RF power amplifier, this technique predistorts the quadrature input signal to compensate for the nonlinearity of the power amplifier. Three implementation algorithms, namely the secant method, the Rotate and Scale method, and the cubic spline interpolation method, have been proposed and studied. Although the performance of these methods satisfies the IS-54 standard, these methods suffer from one or more problems, such as sensitivity to initial conditions, intensive computation requirements that limit the input signal bit rate, and low L/S ratio. In this project, an enhanced linearization method, based on the complex gain predistortion technique, has been investigated and implemented. This method achieves a stable and practicable linearizer with a short LUT update time, a relatively high L/S ratio requiring only a small amount of memory, and moderate computational power. The enhanced linearizer consists of five processing units: a real-time predistortion unit, an off-line distortion estimation unit, a power regulation unit, a clipping control unit, and a loop selector. Most of these units were implemented in a TMS320C30 DSP, with some parts being implemented in an IBM compatible 486 computer.

In order to test the performance of the linearizer, a simplified digital transmitter was constructed. There are four components in the implemented transmitter: the data source, a $\pi/4$ DQPSK encoder and pulse shaping filter, the enhanced linearizer, and the simulated RF power amplifier. The Class B power amplifier characteristic was simulated in two different ways: using digital emulation and using an op-amp diode circuit.

The $\pi/4$ DQPSK encoder and pulse shaping filter were implemented in a TMS320C26 DSP, with some digital circuitry. The test results for this encoder and filter show that the power spectral densities in the adjacent channels are well below the maximum values allowed by the IS-54 standard.

Four sets of tests have been carried out in order to examine the performance of the enhanced linearizer in the simplified digital transmitter. The test results show that the enhanced linearizer has good performance, suppressing the out-of-band spectral distortion by up to 28 dB. In addition, the linearizer achieves a fast LUT updating rate of 33 tables/s, and a short LUT convergence time of ~2 seconds.

As observed in the test results, the performance of the enhanced linearizer is dependent on the distortion mechanism of a power amplifier, the transfer response of the anti-aliasing filter and the reconstruction filter, and the sampling rate of the linearizer. In the enhanced linearization method, the distortion of the amplifier is assumed to be dependent only on the input signal magnitude, i.e., memoryless AM-AM distortion and AM-PM distortion. In this case, any distortion, which does not relate to the input signal magnitude, cannot be corrected by the enhanced linearizer, resulting some distorted output signal. Moreover, since the delay compensator can only correct the constant time delay in the feedback signal, a large phase variation in the anti-aliasing filter and the reconstruction filter causes an additional 10 dB distortion in the output power spectrum density in the adjacent channels. This 10 dB distortion may be reduced by re-designing those filters with a relatively constant phase response. The value of the sampling rate is determined by two factors: the input data bit rate and the distortion characteristic of the amplifier. Increasing the input data bit rate or the distortion of an amplifier cause a wider bandwidth transmitted signal, i.e., signal information occurs at high frequency. Therefore, in order to calculate an accurate predistortion vector, the linearizer requires a higher sampling rate for obtaining the signal information at high frequency. According to the test results, the enhanced linearizer can provide a reasonable correction to the amplifier distortion at a sampling rate of 151.2 ksample/s and an input data bit rate of 18.9 kb/s. In order to achieve the data bit rate of 48.6 kb/s, required by the IS-54 standard, the minimum sampling rate may be as high as 300 ksample/s or 400 ksample/s. Using this high sampling rate, the linearizer needs to be implemented in a faster DSP, such as a TMS320C50 or even TMS320C80.

The following items are suggested for further research:

- In this thesis, the linearizer has only been tested with simulated amplifier characteristics. The performance of the enhanced linearization method needs to be investigated when a real RF power amplifier with a quadrature modulator and demodulator is used.

- Since the performance of the enhanced linearizer has been determined assuming an ideal quadrature modulator and demodulator, it is necessary to implement the compensation algorithm in the enhanced linearizer in order to compensate the impairments of the quadrature modulator and demodulator.

- In the fractional time delay compensator, the fractional time delay coefficients are assumed to be constants, and are obtained by calibration beforehand. It is possible to investigate an algorithm to estimate the time delay coefficients in real time.

- In this project, the enhanced linearizer was implemented in the TMS320C30 DSP development platform and the IBM compatible 486 computer. The enhanced linearizer can be completely implemented in one DSP.

- The linearizer's software for the real-time predistortion unit and the off-line distortion estimation unit can be modified and implemented in a hierarchical multi-processing structure. In this case, several linearizers (i.e., slave DSPs) can be controlled by a master DSP, and share a common linearization object program. As a result, a multichannel transmitter [50] with linearization can be achieved. Figure 6-1 shows a block diagram of such a transmitter.

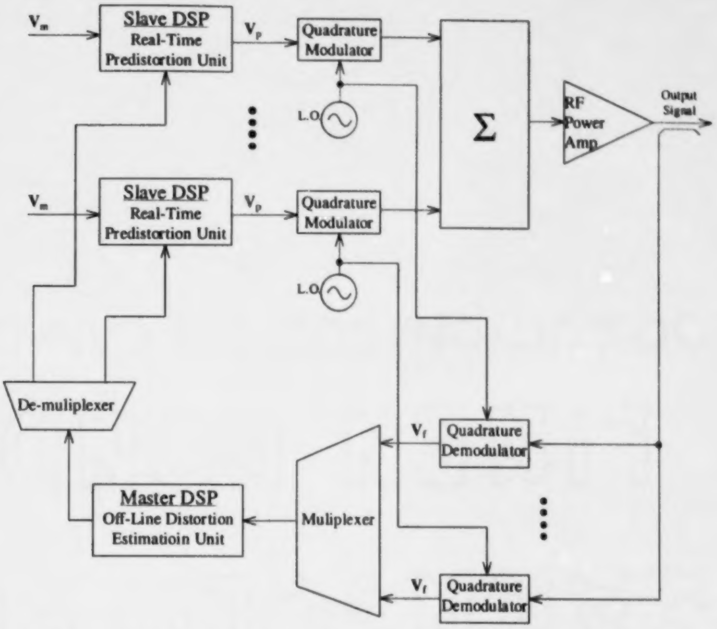

Vm
Slave DSP
Real-Time
Predistortion Unit
Vp
Quadrature
Modulator
L.O
Σ
RF
Power
Amp
Output
Signal
Vm
Slave DSP
Real-Time
Predistortion Unit
Vp
Quadrature
Modulator
L.O
De-muliplexer
Vf
Quadrature
Demodulator
Master DSP
Off-Line Distortion
Estimatioin Unit
Muliplexer
Vf
Quadrature
Demodulator

Figure 6-1. Block diagram of a multichannel transmitter.

REFERENCES

[1] EIA/TIA Interim Standard, Cellular System Dual-Mode Station-Base Station Compatibility Standard, IS-54, *Electronic Industries Association,* May 1990.

[2] S. Chennakeshu, G. J. Saulnier, "Differential Detection of π/4-shifted-DQPSK for Digital Cellular Radio", *Proceedings of the 41st IEEE Vehicular Technology Conference;* 19 - 21 May 1991, St. Louis, MO. New York: IEEE Press, 1991, pp. 186-191.

[3] Y. J. den Otter, "Adaptive Linearization of Power Amplifiers for Digital Cellular Radio", *M. Sc. Thesis,* University of Alberta, 1992.

[4] A. Lohtia, "Linearization of Power Amplifiers", *M. Sc. Thesis,* University of Alberta, Fall 1993.

[5] R. D. Stewart, F. F. Tusubira, "Feedforward Linearization of 950 MHz Amplifiers", *IEE Proceedings,* vol. 135, Pt.H, no.5, pp. 379-350, April 1988.

[6] A. Bateman, D.M. Haines, R.J. Wilkinson, "Linear Transceiver Architectures", *Proceedings of the 38th IEEE Vehicular Technology Conference;* 15-17 June 1988, Philadelphia, PA. New York: IEEE Press 1988, pp. 478-484.

[7] S. A. Hetzel, A. Bateman, J. P. McGeehan, "A LINC Transmitter", *Proceedings of the 41st IEEE Vehicular Technology Conference;* 19-22 May 1991, St. Louis, MO. New York: IEEE Press 1991, pp. 133-137.

[8] A. Bateman and D. M. Haines, "Direct Conversion Transceiver Design for Compact Low-Cost Portable Mobile Radio Terminals", *Proceedings of the 39th IEEE Vehicular Technology Conference;* 1-3 May 1989, San Francisco, CA. New York: IEEE Press, 1989, pp. 57-62.

[9] M. Johansson, T. Mattsson, "Transmitter Linearization Using Cartesian Feedback For Linear TDMA Modulation", *Proceedings of the 41st IEEE Vehicular Technology Conference;* 19 - 21 May 1991, St. Louis, MO. New York: IEEE Press, 1991, pp. 439 - 444.

[10] Y. Akaiwa, Y. Nagata, "Highly Efficient Digital Mobile Communications with a Linear Modulation Method", *IEEE Journal on Selected Areas in Communication,* vol. SAC-5, no. 5, pp. 890-895, June 1987.

[11] S. Ono, N. Kondoh, Y. Shimazaki, "Digital Cellular System with Linear Modulator", *Proceedings of the 39th IEEE Vehicular Technology Conference;* 1-3 May 1989, San Francisco, CA. New York: IEEE Press, 1989, pp. 44-49.

[12] A. A. M. Saleh, J. Salz, "Adaptive Linearization of Power Amplifiers in Digital Radio Systems", *The Bell System Technical Journal,* vol. 62, pp. 1019-1033, April 1983.

[13] A. Lohtia, P. A. Goud and C. G. Englefield, "Power Amplifier Linearization using Cubic Spline Interpolation", *Proceedings of the 43rd IEEE Vehicular Technology Conference;* 1993, Secaucus, NJ, USA: IEEE Press 1993, pp. 676-679.

[14] M. Faulkner, T. Mattsson, W. Yates, "Adaptive Linearization using Pre-Distortion", *Proceedings of the 40th IEEE Vehicular Technology Conference;* 6-9 May 1990, Orlando, FL. NewYork : IEEE Press, 1990, pp. 35-39.

[15] A. Bateman, "The Combined Analogue Locked Loop Universal Modulator (CALLUM)", *Proceedings of the 42nd IEEE Vehicular Technology Conference;* May 1992, Denver, Colorado. New York: IEEE Press, 1992, pp. 759-763.

[16] Y. Nagata, "Linear Amplification Technique for Digital Mobile Communication", *Proceedings of the 39th IEEE Vehicular Technology Conference;* 1-3 May 1989, San Francisco, CA. New York: IEEE Press 1989, pp. 159-164.

[17] J. K. Cavers, "A Linearizing Predistorter With Fast Adaptation", *Proceedings of the 40th IEEE Vehicular Technology Conference;* 6-9 May 1990, Orlando, FL. NewYork : IEEE Press, 1990, pp. 41-47.

[18] J. K. Cavers, "Amplifier Linearization Using A Digital Predistorter with Fast Adaptation and Low Memory Requirements", *IEEE Transactions on Vehicular Technology,* vol. VT-39, no. 4, pp. 374-382, Nov. 1990.

[19] A. S. Wright, W. G. Durtler, "Experimental Performance of an Adaptive Digital Linearized Power Amplifier", *IEEE Transactions on Vehicular Technology,* vol. 41, no. 4, Nov 1992, pp. 395-400.

[20] J. D. Lenk, Practical Guide to Electronic Amplifiers, *Prentice-Hall Inc.,* Englewood Cliffs, New Jersey, 1991.

[21] D. E. Norton, S. A. Massa and P. O'Donovan, "I and Q Modulators for Cellular Communications Systems" *Microwave Journal,* Oct 1991, pp. 63,65,66,70,72,74,77-80.

[22] J. Roome, "Analysis of Quadrature Detectors Using Complex Envelope Notation", *IEE Proceedings,* Vol. 136, Pt.F, No.2,pp. 95-100, April 1989.

[23] K. Anvari, M. Kaube and B. Hriskevich, "Performance of a Direct Conversion Receiver with π/4 DQPSK Modulated Signal", *Proceedings of the 41st IEEE Vehicular Technology Conference;* 19-21 May 1991, St. Louis, MO. New York: IEEE Press, 1991, pp. 822-827

[24] A. E. Jones, J. G. Gardiner, "Phase Error Correcting Vector Modulator For Personal Communications Network (PCN) Transceivers"; *Electronics Letters;* 4 July 1991, vol. 27, No. 14, pp 1230-1231.

[25] A. Lohtia, P. A. Goud and C. G. Englefield, "An Adaptive Digital Technique For Compensating for Analog Quadrature Modulator/Demodulator Impairments", *Proceedings of the IEEE 1993 Pacific Rim Conference on Communication, Computers and Signal Processing;* 1993, Victoria, BC, Can: IEEE Press 1993, pp. 447-450.

[26] M. Faulkner, T. Mattsson, W. Yates, "Automatic Adjustment of Quadrature Modulators", *Electronics Letters;* 31 January 1991, vol. 27, No. 3, pp 214-216.

[27] M. Faulkner, M. Johannsson, "Correction of Mixer Nonlinearity in Quadrature Modulators", *Electronics Letters;* 30 January 1992, vol. 28, No. 3, pp 293-295.

[28] D. Hilborn, S. P. Stapleton and J. K. Cavers, "An Adaptive Direct Conversion Transmitter", *Proceedings of the 42nd IEEE Vehicular Technology Conference;* May 1992, Denver, Colorado. New York: IEEE Press, 1992, pp. 764 - 767.

[29] RF/IF Designer's Guide, SF Summer 94, *Mini-Circuits,* pp. 115-117.

[30] E. A. Lee and D. G. Messerschmitt, "Digital Communication", Kluwer Academic Publishers 1990. Chapter 15, pp. 571-584.

[31] V. N. Yarmolik and S. N. Demidenko, Generation and Application of Pseudorandom Sequences for Random Testing, John Wiley and Sons, Chapters 1 and 3, 1988.

[32] User's Guide For TMS320C2x, *Texas Instruments,* 1993.

[33] S. Kumar, T. Danshin, "A Direct π/4 DQPSK modulator With Memory Based Prefiltering and Predistortion", *1993 IEEE Wescanex;* 17-18 May 1993, Saskatoon, Can: IEEE Press, pp. 208-211.

[34] D. M. Hoover, "An Instrument for Testing North American Digital Cellular Radio", *Hewlett-Packard Journal,* pp. 65-71, April 1991.

[35] Bokulic, R. S., "Design of Differential QPSK Modulator", *Electronics Letters,* Vol. 27, No. 13, pp. 1185-1186, April 1991

[36] Digital Signal Processing Application Seminar, *Analog Devices, Digital Signal Processing Division,* pp. 4.1-4.29, Fall 1991.

[37] Kuc, Roman, Introduction to Digital Signal Processing, *McGraw-Hill Book Company,* 1988.

[38] Microsoft Visual C++ Programmer's Guides, *Microsoft Corporation*, Doc no. DB29682-0193, 1993.

[39] B. Stroustrup, The C++ Programming Language, *Addison Wesley publishing co.*, 2nd ed., 1993.

[40] TMS320C30 System Board User's Manual, *Spectrum Signal Processing Inc*, Doc no. 500-00001, Version 1.02, Oct 1992.

[41] T. Neufeld, "An Adaptive Digital Technique for RF Power Amplifier Linearization", *SummerStudent Report and Presentation*, TRlabs, 1993.

[42] Theory, Algorithms, and Implementations, vol 2, *Texas Instruments*, 1990.

[43] User's Guide For TMS320 Floating-Point DSP Assembly Language Tools, *Texas Instruments*, 1993.

[44] User's Guide For TMS320C3x, *Texas Instruments*, 1990.

[45] User's Guide For TMS320 Floating-Point DSP Optimizing C Compiler, *Texas Instruments*, 1991.

[46] Microsoft Visual C++ Reference Volume II", *Microsoft Corporation*, Doc no. DB35801-0193, 1993.

[47] Microsoft Visual C++ Reference Volume III", *Microsoft Corporation*, Doc no. DB50574-0193, 1993.

[48] B. W. Kernighan, D. M. Ritchie, The C Programming Language, *Prentice-Hall Inc.*, Englewood Cliffs, New Jersey, 1978.

[49] Correspondence with A. Wright, NovAtel, Calgary, AB, 1991, 1992.

[50] M. Johansson, *et al.* "Linearization of Multi-Carrier Power Amplifiers", *Proceedings of the 43rd IEEE Vehicular Technology Conference;* 1993, Secaucus, NJ, USA: IEEE Press 1993, pp. 684-687.

APPENDIX A.
MINIMUM MEAN SQUARE ERROR (MMSE) TIMING RECOVERY ALGORITHM.

The MMSE timing recovery algorithm is used to estimate the time difference between the retrieved clock from a received modulated signal and a reference clock. According to the time difference, the reference clock can be adjusted in order to match the retrieved clock. As a result, the clock of the received modulated signal is recovered. Using the idea of estimating the time difference, the MMSE algorithm can be used to calculate the signal delay of a unknown component, and can easily be implemented in a DSP. The following mathematical derivation is to illustrate how the MMSE algorithm is used to determine the signal delay.

In the general case, the output signal *S* and the delayed received signal $R(\tau)$ are complex signals, where τ is the time delay. The error (*e*) between *S* and $R(\tau)$ is defined:

$$e_i = R_i(\tau) - S_i \tag{1}$$

where *i* is the sampling index.The mean square values of e_i are written as:

$$E\left(\left|e_i\right|^2\right) = E\left(\left|R_i(\tau) - S_i\right|^2\right) \tag{2}$$

where *E*() is the mean value function. In order to find an accurate value for τ, the value of $E(|e_i|^2)$ should be minimized. To obtain the minimum value, the derivative of $E(|e_i|^2)$ is taken.

$$\frac{\partial}{\partial \tau_i} E\left(\left|e_i\right|^2\right) = E\left(\frac{\partial}{\partial \tau_i}\left|e_i\right|^2\right) = E\left(\frac{\partial}{\partial \tau_i}\left|R_i(\tau) - S_i\right|^2\right) \tag{3}$$

Since an analytic solution for (3) is difficult to obtain, an iterative solution may be attempted by iterating the value of τ in the reverse direction of the derivative of $E(|e_i|^2)$.

$$\tau_{i+1} = \tau_i - k \cdot \frac{\partial}{\partial \tau_i}\left|e_i\right|^2 \tag{4}$$

where *k* is the damping ratio in the iteration. The derivative of e_i can be written as:

$$\begin{aligned} \frac{\partial}{\partial \tau_i}\left|e_i\right|^2 &= \frac{\partial}{\partial \tau_i}\left(e_i \cdot e_i^*\right) \\ &= 2 \cdot \mathrm{Re}\left(e_i^* \cdot \frac{\partial e_i}{\partial \tau_i}\right) \end{aligned} \tag{5}$$

Substituting (5) into (4) gives:

$$\tau_{i+1} = \tau_i - k' \cdot \mathrm{Re}\left(\left(R_i(\tau) - S_i\right)^* \cdot \frac{\partial R_i(\tau)}{\partial \tau_i}\right) \tag{6}$$

where *k'* is the damping ratio in the iteration. If S_i and $R_i(\tau)$ are real signals, then (6) can be written as:

$$\tau_{i+1} = \tau_i - k' \cdot \left(R_i(\tau) - S_i\right) \cdot \frac{\partial R_i(\tau)}{\partial \tau_i} \tag{7}$$

From (7), the value of τ can be iterated as soon as the derivative of $R_i(\tau)$ is found. To obatin the derivative of $R_i(\tau)$, the Nyquist sampling theory is applied. The continuous received signal $R(t)$ can be reconstructed by applying an ideal low pass filter to the sampled received signal $R_i(\tau)$.

$$R(t)=\sum_n R_n(\tau)\cdot\frac{\sin\left(\frac{\pi}{T}\cdot(t-n\cdot T-\tau)\right)}{\frac{\pi}{T}\cdot(t-n\cdot T-\tau)} \qquad (8)$$

where T is the sampling time period. Since $R_i(\tau) = R(t)|_{t=i\cdot T+\tau}$, the derivative of Eq. (8) is valid. In this case, the derivative of $R_i(\tau)$ can be written as the convolution of $R_i(\tau)$ with an IIR filter.

$$\begin{aligned}\frac{\partial R_i(\tau)}{\partial\tau}&=\left.\frac{\partial R(t)}{\partial t}\right|_{t=i\cdot T+\tau}\\&=\sum_n R_n(\tau)\cdot\frac{\partial}{\partial t}\left.\left(\frac{\sin\left(\frac{\pi}{T}\cdot(t-n\cdot T-\tau)\right)}{\frac{\pi}{T}\cdot(t-n\cdot T-\tau)}\right)\right|_{t=i\cdot T+\tau} \qquad (9)\\&=R_i(\tau)*d_i\end{aligned}$$

where

$$d_i=\begin{cases}0 & , \quad i=0\\ \frac{(-1)^i}{i\cdot T} & , \quad i\neq 0\end{cases} \qquad (10)$$

The filter d_i can now be approximated by using an N tap direct form FIR filter. Therefore, Eq. (7) may be rewritten as follows:

For N=3,

$$\tau_{i+1} = \tau_i - \frac{k}{T} \cdot \left(R_i(\tau) - S_i\right) \cdot \left(R_{i+1}(\tau) - R_{i-1}(\tau)\right) \tag{11}$$

For N=5

$$\tau_{i+1} = \tau_i - \frac{k}{2 \cdot T} \cdot \left(R_i(\tau) - S_i\right) \cdot \left(R_{i-2}(\tau) - 2 \cdot R_{i-1}(\tau) + 2 \cdot R_{i+1}(\tau) - R_{i+2}(\tau)\right) \tag{12}$$

APPENDIX B.
CIRCUIT DIAGRAMS FOR THE π/4 DQPSK ENCODER AND PULSE SHAPING FILTER.

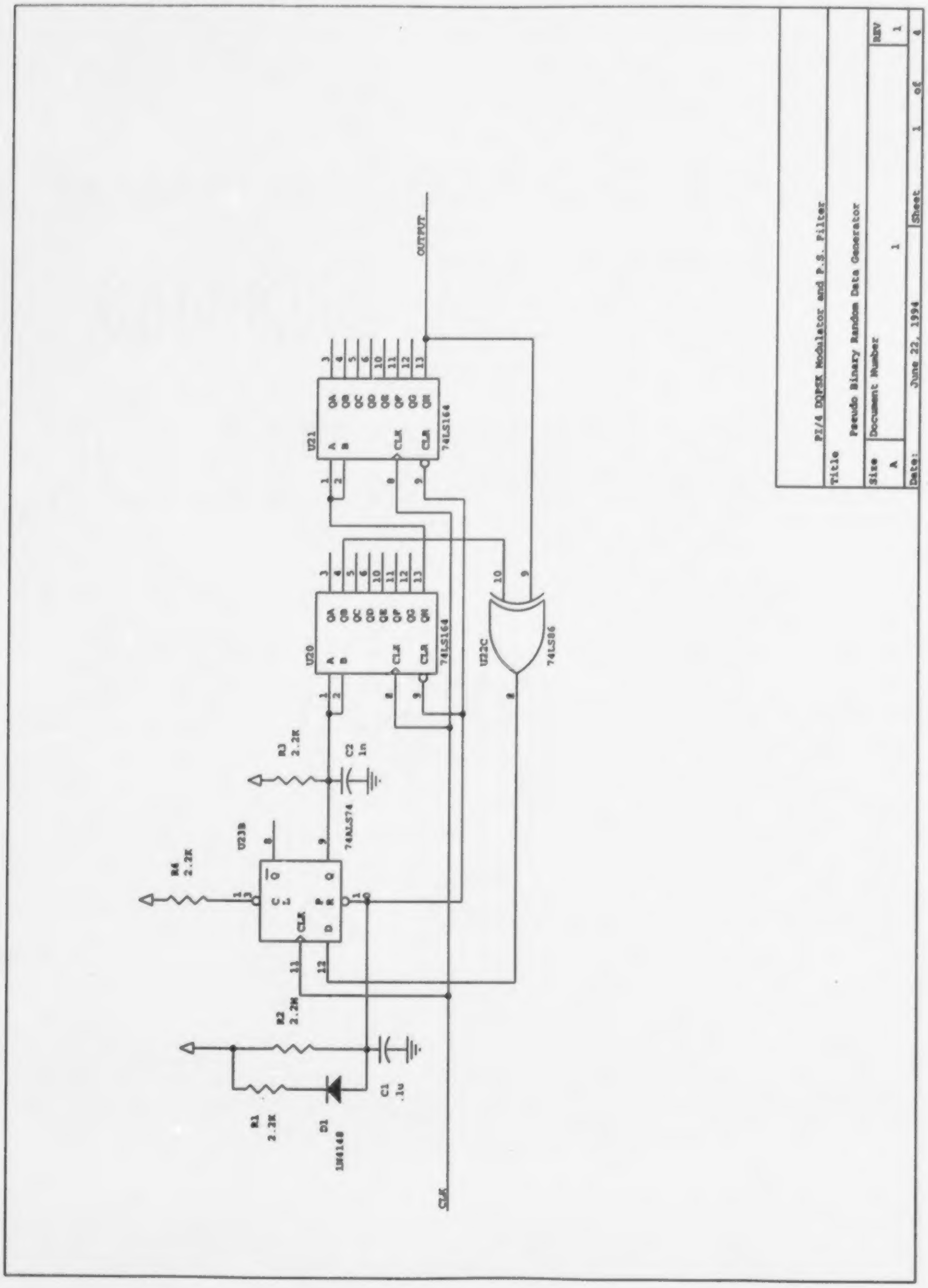

PI/4 DQPSK Modulator and P.S. Filter
Title
Pseudo Binary Random Data Generator
Size
A
Document Number
1
REV
1
Date: June 22, 1994
Sheet 1 of 4
OUTPUT
CLK
U21
74LS164
U20
74LS164
QA QB QC QD QE QF QG QH
A B CLK CLR
U22C
74LS86
U23B
74ALS74
R1 2.2K
R2 2.2M
R3 2.2K
R4 2.2K
C1 .1u
C2 1n
D1 1N4148

Figure B-1. Circuit diagram of the implemented data source.

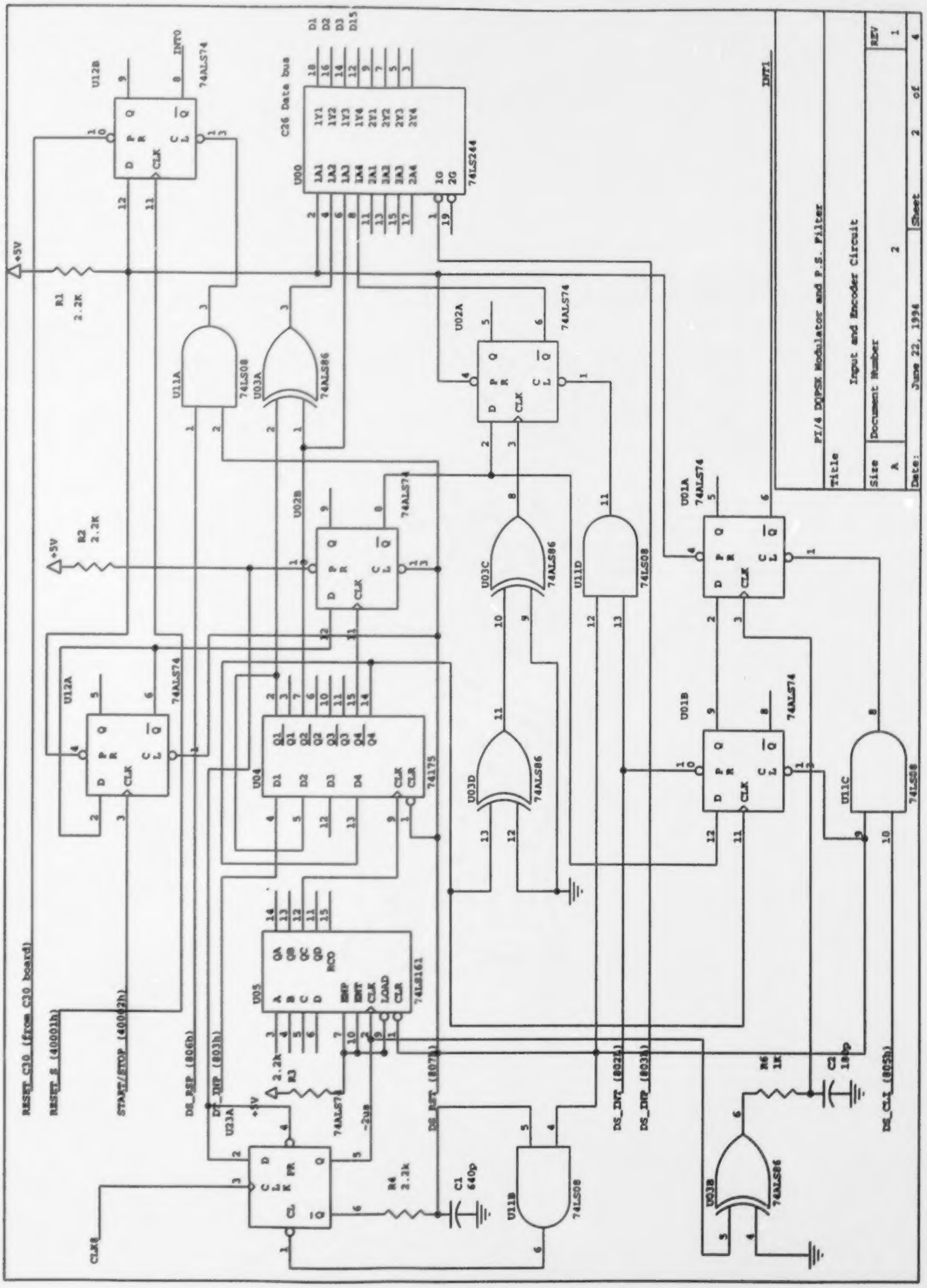

PI/4 DQPSK Modulator and P.S. Filter
Title
Input and Encoder Circuit
Size
A
Document Number
2
REV
1
Date:
June 22, 1994
Sheet
2
of
4
C26 Data bus
D1
D2
D3
D15
INT0
INT1
U00
74LS244
U12B
U12A
74ALS74
U11A
74LS08
U03A
74ALS86
U02A
74ALS74
U02B
U03C
U03D
U11D
U11C
U11B
U01A
U01B
U04
74175
U05
74LS161
U23A
74ALS74
R1
2.2K
R2
2.2K
R3
2.2k
R4
2.2k
R6
1K
C1
640p
C2
100p
+5V
RESET_C30 (from C30 board)
RESET_S (4000lh)
START/STOP (40002h)
DS_RSP (806h)
DT_INP (803h)
DS_RST (807h)
DS_INT (802h)
DS_INP (803h)
DS_CLK (805h)
CLK8
~2us

Figure B-2. Circuit diagram of the π/4 DQPSK encoder circuit

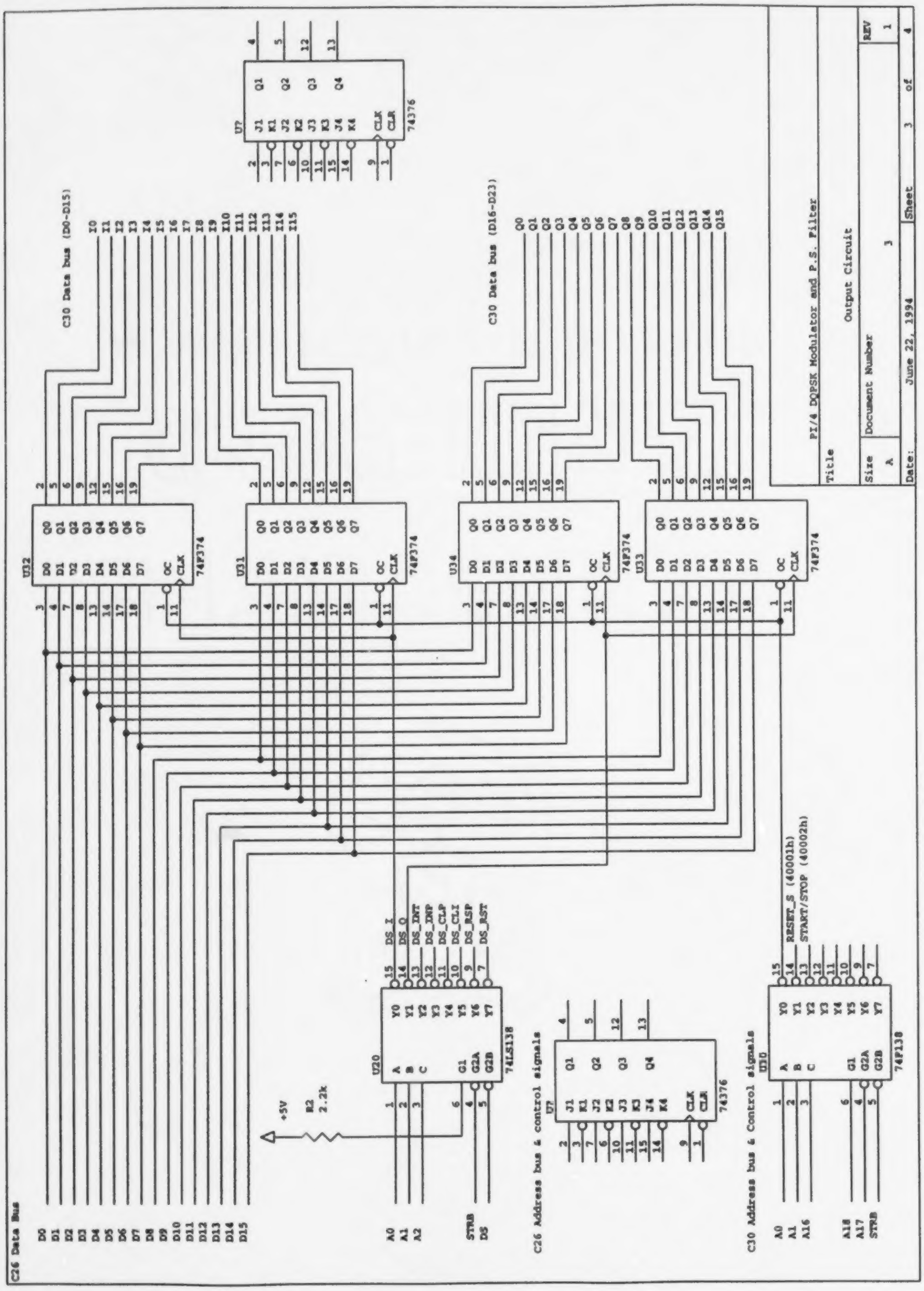

PI/4 DQPSK Modulator and P.S. Filter
Title
Output Circuit
Size
A
Document Number
3
REV
1
Date: June 22, 1994
Sheet 3 of 4
C26 Data Bus
C30 Data bus (D0-D15)
C30 Data bus (D16-D23)
U32
U31
U34
U33
74F374
U7
74376
U20
74LS138
U30
74F138
R2
2.2k
+5V
DS_I
DS_Q
DS_INT
DS_DNP
DS_CLP
DS_CLI
DS_RSP
DS_RST
RESET_S (40001h)
START/STOP (40002h)
C26 Address bus & control signals
C30 Address bus & Control signals
STRB
DS

Figure B-3. Circuit diagram of the digital pulse shaping filter circuit

APPENDIX C.
NORMALIZED AM-AM AND AM-PM CONVERSION FACTORS FOR THE CLASS B AMPLIFIER, AND OP-AMP DIODE CIRCUIT.

Table C-1. TRANSFER CHARACTERISTICS OF A CLASS B RF POWER AMPLIFIER.

Input	Output	Phase (degrees)
0.30	0.3078	1.1364
0.31	0.3200	1.4085
0.32	0.3321	1.6790
0.33	0.3441	1.9216
0.34	0.3563	2.0990
0.35	0.3688	2.1763
0.36	0.3818	2.1212
0.37	0.3956	1.9220
0.38	0.4091	1.6864
0.39	0.4215	1.5497
0.40	0.4329	1.5400
0.41	0.4443	1.6338
0.42	0.4566	1.8116
0.43	0.4693	2.0681
0.44	0.4817	2.4015
0.45	0.4937	2.7942
0.46	0.5056	3.2149
0.47	0.5178	3.6355
0.48	0.5304	4.0402
0.49	0.5431	4.4176
0.50	0.5559	4.7572
0.51	0.5685	5.0489
0.52	0.5809	5.2877
0.53	0.5930	5.4803
0.54	0.6049	5.6342
0.55	0.6166	5.7567
0.56	0.6280	5.8550
0.57	0.6393	5.9355
0.58	0.6503	6.0046
0.59	0.6611	6.0681
0.60	0.6717	6.1317
0.61	0.6823	6.2006
0.62	0.6924	6.2797
0.63	0.7024	6.3699
0.64	0.7124	6.4691
0.65	0.7221	6.5755
0.66	0.7318	6.6973
0.67	0.7412	6.8027
0.68	0.7506	6.9203
0.69	0.7598	7.0384
0.70	0.7689	7.1555
0.71	0.7778	7.2703
0.72	0.7867	7.3814
0.73	0.7954	7.4875
0.74	0.8041	7.5882
0.75	0.8126	7.6838
0.76	0.8211	7.7750
0.77	0.8294	7.8622
0.78	0.8377	7.9459
0.79	0.8459	8.0267
0.80	0.8539	8.1050
0.81	0.8619	8.1811
0.82	0.8699	8.2557
0.83	0.8777	8.3290
0.84	0.8855	8.4014
0.85	0.8932	8.4732
0.86	0.9008	8.5445
0.87	0.9084	8.6153
0.88	0.9159	8.6855
0.89	0.9233	8.7550
0.90	0.9306	8.8239
0.91	0.9379	8.8922
0.92	0.9451	8.9597
0.93	0.9522	9.0265
0.94	0.9592	9.0925
0.95	0.9662	9.1578
0.96	0.9731	9.2222
0.97	0.9800	9.2859
0.98	0.9867	9.3487
0.99	0.9934	9.4106
1.00	1.0000	9.4717

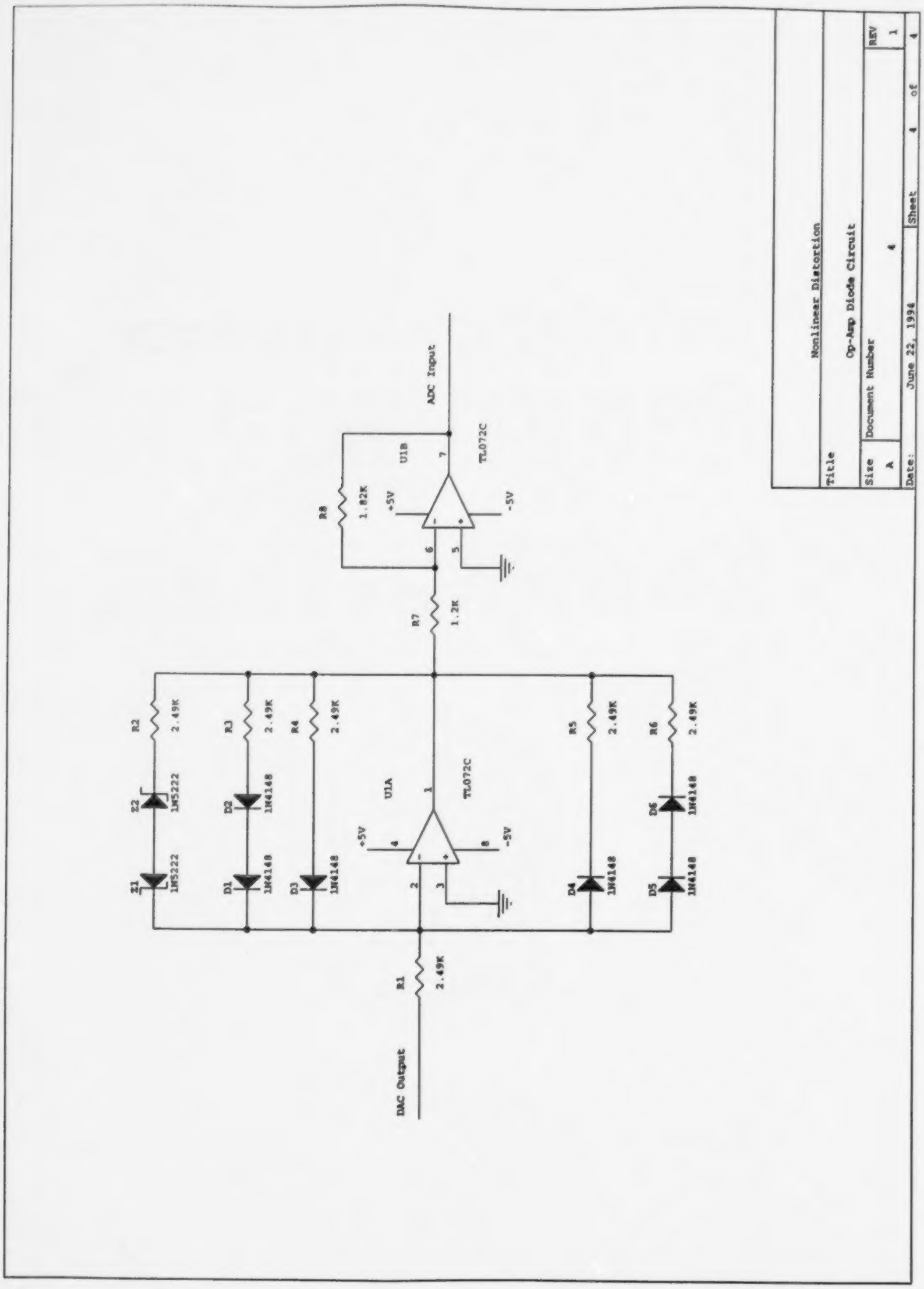
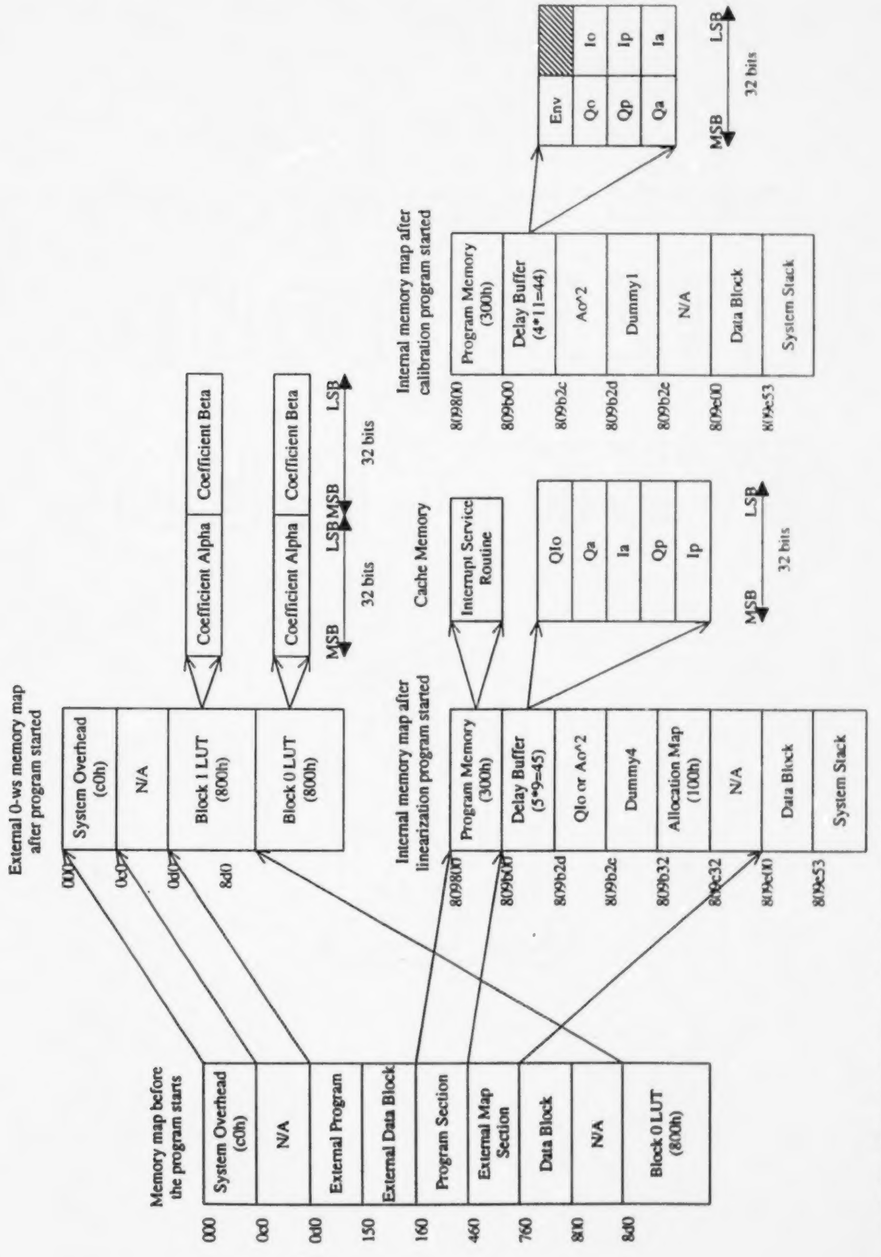

DAC Output
R1
2.49K
R2
2.49K
Z1
1N5222
Z2
1N5222
R3
2.49K
D1
1N4148
D2
1N4148
R4
2.49K
D3
1N4148
U1A
TL072C
+5V
-5V
1
2
3
4
8
R5
2.49K
D4
1N4148
R6
2.49K
D5
1N4148
D6
1N4148
R7
1.2K
R8
1.82K
U1B
TL072C
+5V
-5V
5
6
7
ADC Input
Nonlinear Distortion
Title
Op-Amp Diode Circuit
Size
A
Document Number
4
REV
1
Date:
June 22, 1994
Sheet
4
of
4

Figure C-1. Circuit diagram of the base-band op-amp diode circuit

APPENDIX D.
MEMORY MAP FOR THE LINEARIZER DSP PROGRAMS.


Memory map before the program starts
000
0c0
0d0
150
160
460
760
800
8d0
System Overhead (c0h)
N/A
External Program
External Data Block
Program Section
External Map Section
Data Block
N/A
Block 0 LUT (800h)
External 0-ws memory map after program started
000
0c0
0d0
8d0
System Overhead (c0h)
N/A
Block 1 LUT (800h)
Block 0 LUT (800h)
Coefficient Alpha
Coefficient Beta
MSB
LSB
32 bits
Internal memory map after linearization program started
809800
809b00
809b2d
809b2e
809b32
809e32
809e00
809e53
Program Memory (300h)
Delay Buffer (5*9=45)
QIo or Ao^2
Dummy4
Allocation Map (100h)
N/A
Data Block
System Stack
Cache Memory
Interrupt Service Routine
QIo
Qa
Ia
Qp
Ip
Internal memory map after calibration program started
809800
809b00
809b2c
809b2d
809b2e
809e00
809e53
Program Memory (300h)
Delay Buffer (4*11=44)
Ao^2
Dummy1
N/A
Data Block
System Stack
Env
Qo
Io
Qp
Ip
Qa
Ia

Note: In the delay buffer, the first data set of Ao^2 is corresponding to the second data set since there is one time slot delay

Figure D-1. Memory map for the internal memory and the external 0ws memory.

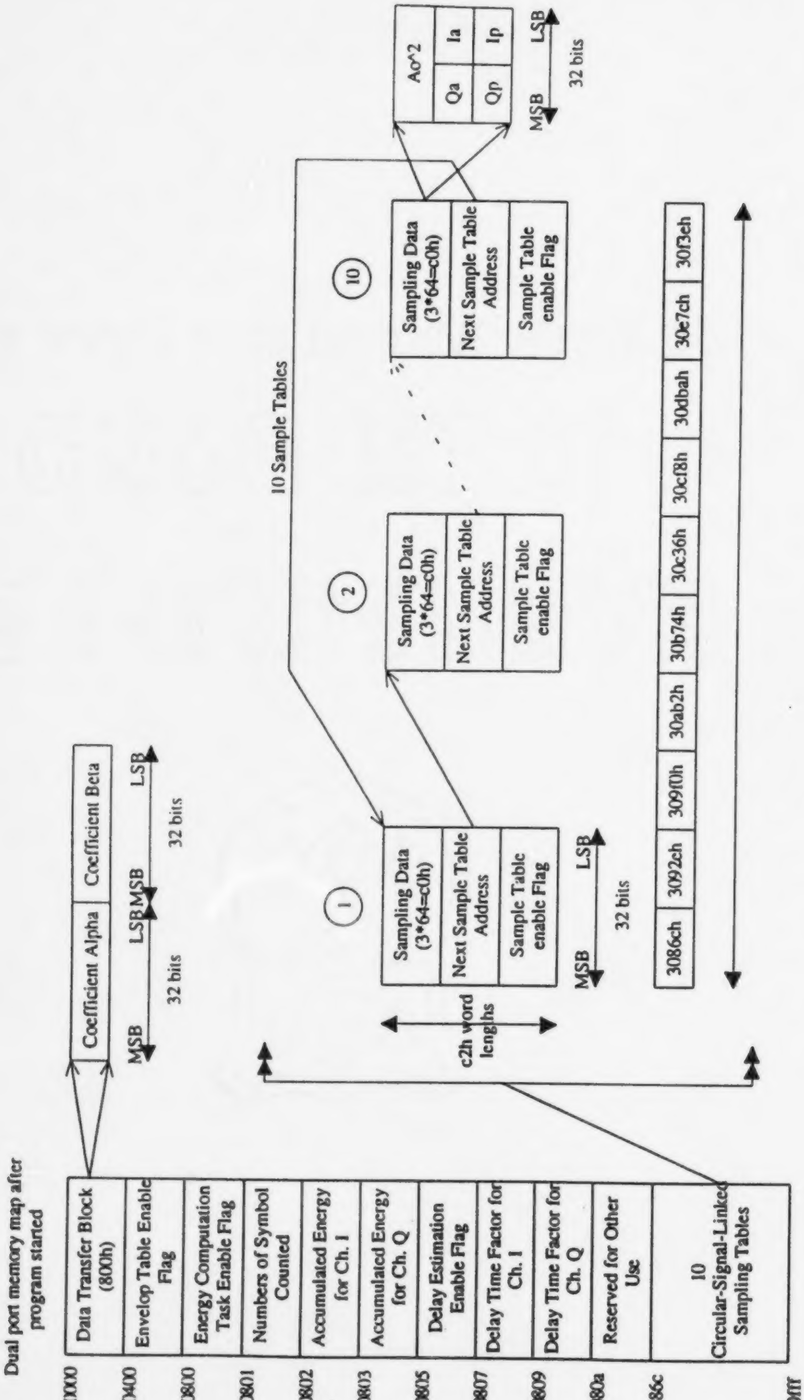

Dual port memory map after program started
30000
30400
30800
30801
30802
30803
30805
30807
30809
3080a
3086c
30fff
Data Transfer Block (800h)
Envelop Table Enable Flag
Energy Computation Task Enable Flag
Numbers of Symbol Counted
Accumulated Energy for Ch. I
Accumulated Energy for Ch. Q
Delay Estimation Enable Flag
Delay Time Factor for Ch. I
Delay Time Factor for Ch. Q
Reserved for Other Use
10 Circular-Signal-Linked Sampling Tables
Coefficient Alpha
Coefficient Beta
MSB
LSB
MSB
LSB
32 bits
32 bits
c2h word lengths
①
Sampling Data (3*64=c0h)
Next Sample Table Address
Sample Table enable Flag
MSB
LSB
32 bits
②
Sampling Data (3*64=c0h)
Next Sample Table Address
Sample Table enable Flag
10 Sample Tables
⑩
Sampling Data (3*64=c0h)
Next Sample Table Address
Sample Table enable Flag
Ao^2
Qa
Ia
Qp
Ip
MSB
LSB
32 bits
3086ch
3092eh
309f0h
30ab2h
30b74h
30c36h
30cf8h
30dbah
30e7ch
30f3eh

Figure D-2. Memory map for the external dual-port 2ws memory.

APPENDIX E.

PROGRAMMING LISTS FOR THE π/4 DQPSK ENCODER AND PULSE SHAPING FILTER, AND THE ENHANCED LINEARIZER.

LIST	Content	Page
1	TMS320C26 DSP program file for the π/4 DQPSK encoder and pulse shaping filter. File Name: D_syn.asm	130
2	TMS320C30 DSP memory block definition file for the enhanced linearizer and pre-calibration DSP program. File Name: map.cmd	135
3	TMS320C30 DSP macro function file for the enhanced linearizer and pre-calibration DSP program. File Name: Macro.fcn	136
4	TMS320C30 DSP constant and symbol header file for the enhanced linearizer and pre-calibration DSP program. File Name: Const.h30	138
5	TMS320C30 DSP interrupt vectors header file for the enhanced linearizer and pre-calibration DSP program. File Name: int_vec.h30	142
6	TMS320C30 DSP memory allocation header file for the enhanced linearizer and pre-calibration DSP program. File Name: Memory.h30	143
7	TMS320C30 DSP initialization file for the enhanced linearizer and pre-calibration DSP program. File Name: Initial.c30	146
8	TMS320C30 DSP pre-calibration program. File Name: 30c.c30	148
9	TMS320C30 DSP enhanced linearizer program. File Name: 30p.c30	154
10	486 PC C++ header files for the enhanced linearizer. File Name: Format.h, Const.h, Cinterfa.h, Clinearl.h, Clineari.h, Ctest0.h, Ctest1.h, Utility.h, Map_cntr.h.	161
11	486 PC C++ program files for the enhanced linearizer. File Name: Cinterfa.cpp.	170

12 486 PC C++ program files for the enhanced linearizer. File Name: Clinearl.cpp. 182

13 486 PC C++ program files for the enhanced linearizer. File Name: Clineari.cpp. 190

14 486 PC C++ program files for the enhanced linearizer. File Name: Ctiming.cpp. 194

15 486 PC C++ program files for the enhanced linearizer. File Name: Ctest0.cpp. 195

16 486 PC C++ program files for the enhanced linearizer. File Name: Ctest1.cpp. 197

17 486 PC C++ program files for the enhanced linearizer. File Name: Utility.cpp. 199

18 486 PC C++ program files for the enhanced linearizer. File Name: Run.cpp. <<Main Program>> 203

19 TMS320C30 DSP digitally emulated power amplifier program. 204

List 1: TMS320C26 DSP program file for the π/4 DQPSK encoder and pulse shaping filter.
File Name: D_syn.asm

```
**************************************************
*
*        D_SYN.ASM
*        Max bit rate: 25kbps
*        occupied two interrupts
*             INT0 : external reset request
*             INT1 : output pulse shaping request
*
*************************************************
TBL             .set          1536               ; dma = 0600h
RND_UP          .set          1539               ; dma = 0603h
TMP_ACC         .set          1540               ; dma = 0604h
NEW_Q           .set          1542               ; dma = 0606h
TBL_R           .set          1551               ; dma = 060fh
NEW_I           .set          1553               ; dma = 0611h

R_IMR           .set          4                  ; dma = 0004h
R_I             .set          0                  ; xdma = 0800h
R_Q             .set          1                  ; xdma = 0801h
DS_INT          .set          2                  ; xdma = 0802h
DS_INP          .set          3                  ; xdma = 0803h
DS_CLP          .set          4                  ; xdma = 0804h
DS_CLI          .set          5                  ; xdma = 0805h
DS_RSP          .set          6                  ; xdma = 0806h
DS_RST          .set          7                  ; xdma = 0807h
MASK_A          .set          64512              ; pma = 0fc00h
GRP0_A          .set          64560              ; pma = 0fc30h
GRP1_A          .set          64568              ; pma = 0fc38h
GRP2_A          .set          64576              ; pma = 0fc40h
GRP3_A          .set          64584              ; pma = 0fc48h
GRP4_A          .set          64592              ; pma = 0fc50h
GRP5_A          .set          64600              ; pma = 0fc58h
GRP6_A          .set          64608              ; pma = 0fc60h
GRP7_A          .set          64616              ; pma = 0fc68h
GRP8_A          .set          64624              ; pma = 0fc70h
GRP9_A          .set          64632              ; pma = 0fc78h
GRP10_A         .set          64640              ; pma = 0fc80h
GRP11_A         .set          64648              ; pma = 0fc88h
GRP12_A         .set          64656              ; pma = 0fc90h
GRP13_A         .set          64664              ; pma = 0fc98h
GRP14_A         .set          64672              ; pma = 0fca0h
GRP15_A         .set          64680              ; pma = 0fca8h

                .ds           0400h
                .data

*************************************************************
*
* Create a table in data memory. Starting address = 600h
*
*               +-----------------+
*               | dma  | data     |
*               +------+----------+
*               | 600  | MASK     |
*               | 601  | PRV_PHA  |
*               | 602  | MOD_8    |
*               | 603  | RND_UP   |
*               | 604  | TMP_ACC  |
*               | 605  | 0000     | <-- used to set P reg to 0
*               | 606  | Q0       | <==> NEW_Q
*               | 607  | Q1       |
*               | 608  | Q2       |
*               | 609  | Q3       |
*               | 60a  | Q4       |
*               | 60b  | Q5       |
*               | 60c  | Q6       |
*               | 60d  | Q7       |
*               | 60e  | DUMMY1   |
*               | 60f  | RND_UP   |
*               | 610  | 0000     | <-- used to set P reg to 0
*               | 611  | I0       | <==> NEW_I
*               | 612  | I1       |
*               | 613  | I2       |
*               | 614  | I3       |
*               | 615  | I4       |
*               | 616  | I5       |
*               | 617  | I6       |
*               | 618  | I7       | <-- TBL_IQ
*               | 619  | DUMMY2   |
*               +-----------------+
*
*************************************************************
```

```
MASK            .word           0eh                 ; Masking the input data
PRV_PHA         .word           0fc20h              ; Initializing the previous ref
                                                    ; phase to state 0 ( same as
                                                    ; TBL_UC addr )
MOD_8           .word           0ffeeh              ; Generating the modulus 8 address
                                                    ; for table lookup
RND             .word           02000h              ; Round up
DATA            .word           0                   ; TMP_ACC
ALL_0           .word           0                   ; <- 0
                .word           0                   ; <- Q0
                .word           0                   ; <- Q1
                .word           0                   ; <- Q2
                .word           0                   ; <- Q3
                .word           0                   ; <- Q4
                .word           0                   ; <- Q5
                .word           0                   ; <- Q6
                .word           0                   ; <- Q7
DUMMY1          .word           0
                .word           02000h              ; Round up
                .word           0                   ; <- 0
                .word           0                   ; <- I0
                .word           0                   ; <- I1
                .word           0                   ; <- I2
                .word           0                   ; <- I3
                .word           0                   ; <- I4
                .word           0                   ; <- I5
                .word           0                   ; <- I6
                .word           0                   ; <- I7
DUMMY2          .word           0

                .ds             0420h
;
; <<<< Table for converting the current phase location to the
;               corresponding unit circle values >>>>
; There are eight different possible states on a unit circle:
;               000,001,010,011,100,101,110,111.
;               Note: This table is located in the program memory
;
; These values are multiplied by the normalize coefficient No=.731134811906 and
; the adjusting coefficient x=.948903417233
; No = 16384 / 22409
; x = sqrt(3ffh/(3.16/3*16384)^2*2^18)
;
TBL_UC          .int            16107,0             ; State 000 = (1 0)
                .int            11389,11389         ; State 001 = (.70711 .70711)
                .int            0,16107             ; State 010 = (0 1)
                .int            -11389,11389        ; State 011 = (-.70711 .70711)
                .int            -16107,0            ; State 100 = (-1 0)
                .int            -11389,-11389       ; State 101 = (-.70711 -.70711)
                .int            0,-16107            ; State 110 = (0 -1)
                .int            11389,-11389        ; State 111 = (.70711 -.70711)

;
; <<<< Table for the pulse shaping filter >>>>
;
; It is divided into sixteen groups, and each group is seperated
; by one of the sixteenth sampling time. Each point within a group
; is seperated by a unit time of symbols
;
;               Note: This table is also located at program memory
;

H_GRP0          .int            -417,936,-1388,17951,-1388,936,-417,33
H_GRP1          .int            -380,1078,-2111,17803,-463,716,-425,92
H_GRP2          .int            -320,1143,-2631,17365,654,421,-399,145
H_GRP3          .int            -243,1138,-2953,16650,1946,58,-339,188
H_GRP4          .int            -156,1071,-3091,15682,3389,-362,-242,218
H_GRP5          .int            -67,952,-3061,14489,4952,-821,-112,232
H_GRP6          .int            19,795,-2887,13109,6596,-1299,47,226
H_GRP7          .int            95,613,-2596,11583,8280,-1773,227,201
H_GRP8          .int            157,420,-2215,9958,9958,-2215,420,157
H_GRP9          .int            201,227,-1773,8280,11583,-2596,613,95
H_GRP10         .int            226,47,-1299,6596,13109,-2887,795,19
H_GRP11         .int            232,-112,-821,4952,14489,-3061,952,-67
H_GRP12         .int            218,-242,-362,3389,15682,-3091,1071,-156
H_GRP13         .int            188,-339,58,1946,16650,-2953,1138,-243
H_GRP14         .int            145,-399,421,654,17365,-2631,1143,-320
H_GRP15         .int            92,-425,716,-463,17803,-2111,1078,-380

                .ps             0fa02h

                B               0fcf8h    ;ISR_0 for reset request
                B               0fcf0h    ;ISR_1 for data input request

                .ps             0fb00h

*
*               Program Initialization
*

START           CONF            2                   ; Set the memory map to config. 2
                LDPK            0                   ; Initial the interrupt mask register
                LAC             R_IMR
                ORK         03h
                SACL        R_IMR

RESTART LDPK    16          ; Set the data page register to 0800h
                LAC         DS_RST,0            ; Reset the external circuit

                LARP        AR2                 ; Create a table in data memory
                LRLK        AR2,TBL
                RPTK        25
                BLKP        MASK_A,*+

                SPM         0                   ; Set no shift in the product shift mode
                SOVM                            ; Set the overflow mode
                SSXM                            ; Set sign extentsion mode
                MPYK        0                   ; Set P reg = 0
                LARK        AR0,21              ; Initial the AR0-6 registers
                LRLK        AR2,TBL
                LRLK        AR3,NEW_I
                LRLK        AR4,NEW_Q
                LRLK        AR6,TBL_R

LOOP0           LAC         DS_INP              ; Get the first symbol
                BLZ         LOOP0               ; Test the input symbol
                SACL        DS_INT              ; Clear the input request
                AND         *+                  ; MASK : Masking the unwanted bits
                ADDS        *+                  ; PRV_PHA : \To calculate the current phase
                AND         *-                  ; MOD_8   : /location wrt the previous one
                SACL        *-,0,AR3            ; prv_PHA : Backup the current phase for ref.
                RPTK        1                   ; Get the unit circle values corresponding
                TBLR        *,AR4               ;         to the current phase loacation

                LARK        AR7,16              ; Set up the loop counter values
LOOP1           IDLE                            ; Wait for the next symbol
                LARP        AR7                 ; i.e. after 16 clock_8 cycles
                SBRK        1
                BANZ        LOOP1

                LRLK        AR5,RND_UP          ; Inital AR5        register
                B           CON0                ; Start the convolution program

*********************************************************************
*                                                                   *
* Convolve the accumulated sybmols with the pulse shape impusle response *
*                                                                   *
*********************************************************************

CON1            IDLE                            ; Wait for symbol sychnous signa
                RPTK        8                   ; Calculate the I values
                MAC         GRP1_A,*-
                SACH        R_I,2               ; Output the result to the external latch
                LAC         *-                  ; Initial ACC values
                SBRK        1                   ; Set the epointer to next table location
                RPTK        8                   ; Calucate the Q values
                MAC         GRP1_A,*-
                SACH        R_Q,2               ; Output the result to the external latch
                LAC         DS_INP              ; Get the input symbls
                SACL        *-                  ; Store it into a specific memory location
                SACL        DS_INT              ; Clear the input request

CON2            IDLE                            ; Same as CON1
                RPTK        8
                MAC         GRP2_A,*-
                SACH        R_I,2
                LAC         *-
                SBRK        1
                RPTK        8
                MAC         GRP2_A,*-
                SACH        R_Q,2
                LAC         *,0,AR2             ; Partially modulate the previous input
                AND         *+,AR5              ; symbol to the PI/4 DQPSK signal
                SACL        *-

CON3            IDLE                            ; Same as CON2
                RPTK        8
                MAC         GRP3_A,*-
                SACH        R_I,2
                LAC         *-
                SBRK        1
                RPTK        8
                MAC         GRP3_A,*-
                SACH        R_Q,2
                LAC         *,0,AR2
                ADDS        *+,AR5
                SACL        *-

CON4            IDLE                            ; Same as CON2
                RPTK        8
                MAC         GRP4_A,*-
                SACH        R_I,2
                LAC         *-
                SBRK        1
                RPTK        8
                MAC         GRP4_A,*-
                SACH            R_Q,2
                LAC             *,0,AR2
                AND             *-,AR5
                SACL            *-

CON5            IDLE                                    ; Same as CON2
                RPTK            8
                MAC             GRP5_A,*-
                SACH            R_I,2
                LAC             *-
                SBRK            1
                RPTK            8
                MAC             GRP5_A,*-
                SACH            R_Q,2
                LAC             *-,0,AR2
                SACL            *-

CON6            IDLE                                    ; Same as CON1
                RPTK            8
                MAC             GRP6_A,*-
                SACH            R_I,2
                LAC             *-
                SBRK            1
                RPTK            8
                MAC             GRP6_A,*-
                SACH            R_Q,2
                SBRK            1                       ; Set AR5 pointer to the next location

CON7            IDLE                                    ; Same as CON6
                RPTK            8
                MAC             GRP7_A,*-
                SACH            R_I,2
                LAC             *-
                SBRK            1
                RPTK            8
                MAC             GRP7_A,*-
                SACH            R_Q,2
                SBRK            1

CON8            IDLE                                    ; Same as CON6
                RPTK            8
                MAC             GRP8_A,*-
                SACH            R_I,2
                LAC             *-
                SBRK            1
                RPTK            8
                MAC             GRP8_A,*-
                SACH            R_Q,2
                SBRK            1

CON9            IDLE                                    ; Same as CON6
                RPTK            8
                MAC             GRP9_A,*-
                SACH            R_I,2
                LAC             *-
                SBRK            1
                RPTK            8
                MAC             GRP9_A,*-
                SACH            R_Q,2
                SBRK            1

CON10           IDLE                                    ; Same as CON6
                RPTK            8
                MAC             GRP10_A,*-
                SACH            R_I,2
                LAC             *-
                SBRK            1
                RPTK            8
                MAC             GRP10_A,*-
                SACH            R_Q,2
                SBRK            1

CON11           IDLE                                    ; Same as CON6
                RPTK            8
                MAC             GRP11_A,*-
                SACH            R_I,2
                LAC             *-
                SBRK            1
                RPTK            8
                MAC             GRP11_A,*-
                SACH            R_Q,2
                SBRK            1

CON12           IDLE                                    ; Same as CON6
                RPTK            8
                MAC             GRP12_A,*-
                SACH            R_I,2
                LAC             *-
                SBRK            1
                RPTK            8
                MAC             GRP12_A,*-
                SACH            R_Q,2
                SBRK            1
```

```
CON13           IDLE                                ; Same as CON6
                RPTK          8
                MAC           GRP13_A,*-
                SACH          R_I,2
                LAC           *-
                SBRK          1
                RPTK          8
                MAC           GRP13_A,*-
                SACH          R_Q,2
                SBRK          1

CON14           IDLE                                ; Same as CON6
                RPTK          8
                MAC           GRP14_A,*-
                SACH          R_I,2
                LAC           *-
                SBRK          1
                RPTK          8
                MAC           GRP14_A,*-
                SACH          R_Q,2
                SBRK          1

CON15           IDLE                                ; Same as CON6
                RPTK          8
                MACD          GRP15_A,*-
                SACH          R_I,2
                LAC           *-
                SBRK          1
                RPTK          8
                MACD          GRP15_A,*-
                SACH          R_Q,2
                LAC           *-,0,AR3              ; Update the table
                RPTK          1
                TBLR          *,AR4

CON0            IDLE                                ; Same as CON6
                RPTK          8
                MAC           GRP0_A,*-
                SACH          R_I,2
                LAC           *-
                SBRK          1
                RPTK          8
                MAC           GRP0_A,*-
                SACH          R_Q,2
                LAC           *-
                B             CON1

                .ds           04f0h
                .data

INT_1           SACL          DS_CLI                ; Clear the interrupt request
                NOP
                NOP
                LARP          AR5                   ; Initial the AR5 pointer and
                LAC           *0+,0                 ; the ACC
                EINT                                ; Enable the interrupt request
                RET

                .ds           04f8h

INT_0           SACL          DS_RSP                ; Clear the interrupt request
                NOP
                NOP
                LALK          RESTART               ; Set the return address
                PUSH
                EINT                                ; Enable the interrupt request
                RET
                .end
```

List 2: TMS320C30 DSP memory block definition file for the enhanced linearizer and pre-calibration DSP program.
File Name: map.cmd

```
/**********************************************************************
*                                                                     *
* File:    map.cmd                                                    *
* Use:     'C30 Linker Command File containing Memory Map             *
*                                                                     *
**********************************************************************/

/* All Memory is RAM */

MEMORY              /* This describes the hardware */
{
/* External SRAM on the Main Board */

          VECTS:    origin=000000h      length=000C0h       /* Part of Bank0 */
          EPROG:    origin=0000d0h      length=00080h       /* Zero-wait */
          EDTBLK:   origin=000150h      length=00010h       /* Zero-wait */

          PROG:     origin=000160h      length=00300h       /* Zero-wait */
          MAP:      origin=000460h      length=00200h       /* Zero-wait */
          DTBLK:    origin=000660h      length=001A0h       /* Zero-wait */

/* the gap in between PROG DTBLK is used for buffering
   since this ragion of memory will be transferred to
   809800h, this gap is reserved for the buffer */

          BANK0:    origin=000800h      length=0F800h       /* Zero-wait */
          BANK1:    origin=010000h      length=10000h       /* SRAM upgrade option */
          BANK2:    origin=020000h      length=10000h       /* SRAM upgrade option */
          BANK3:    origin=030000h      length=0F400h       /* One-wait */

/* Bank 3 is dual-ported between the 'C30 and the PC.
   The length shown is for the default 64Kx4 devices, but 16Kx4 can be used.
   In both cases the top C00h locations are reserved for monitor use.
   If the monitor is not used, you can have this area. */

/* Cached DRAM Memory Expansion on Daughter Board */

          EXPAND:   origin=400000h      length=400000h      /* One of various options */

/* On-chip */

          BLK0_P:   origin=809800h      length=300h
          BLK0_M:   origin=809B00h      length=300h
          BLK1_D:   origin=809E00h      length=200h
}

SECTIONS  /* Allocates uses to the hardware */
{
          .init:    {}        load=VECTS
          .etext:   {}        load=EPROG
          .edata:   {}        load=EDTBLK
          .text:    {}        load=PROG,          run=0809800h
          .map:     {}        load=MAP,           run=0809B00h
          .data:    {}        load=DTBLK,         run=0809d00h
          .dual:    {}        load=BANK3
}
```

List 3: TMS320C30 DSP macro function file for the enhanced linearizer and pre-calibration DSP program.

File Name: Macro.fcn

```
* Generate a single link sampling table
*              SSTBL(no_of_table, size_of_data_set, size_of_table)
*
SLSTBL         .MACRO         no_of_table,size_of_data_set,size_of_table
** Define and initialize two index variables
               .var           indx,next
               .asg           0,indx
               .asg           1,next
               .loop          no_of_table
** next = next % no_of_table
               .if            (next>=no_of_table)
               .eval          next-no_of_table,next
               .endif
** Generate a sampling table
STBL:indx:     .space         size_of_data_set*size_of_table
NEXT:indx:     .word          STBL:next:
STEN:indx:     .space         1
** Increase index variables by 1
               .eval          next+1,next
               .eval          indx+1,indx
               .endloop
               .endm
*
* Insert the number(s) of "LDIP *+objtmpa(nxt_st),objtmpa" command into the program. The number of this
*              command is depended on the numbers of sampling table in the single link table.
*              AHEAD(numbers_of_sampling_table)
*
AHEAD          .MACRO         numbers_of_sampling_table

               .if            (numbers_of_sampling_table<6)
               .emsg          "ERROR!!, The number of sampling table is less than 6"
               .mexit
               .else
               .eval          (numbers_of_sampling_table/2)-2,numbers_of_sampling_table
               .loop          numbers_of_sampling_table
** Where objtmpa and nxt_st are defined in the program
               LDIP           *+objtmpa(nxt_st),objtmpa
               .endloop
               .endif
               .endm
*
* Add one to the specific temporately memory location.
*              ADDONE(memory_location)
*
ADDONE         .MACRO         memory_location
               .var           count,list,tmp_reg
               .asg           0,count
               .asg           "TEMP0, TEMP1, TEMP2, TEMP3, TEMP4, TEMP5, TEMP6, TEMP7",list
               .loop
               .break         ($ismember(tmp_reg,list)==0)
               .if            (memory_location==count)
               .break
               .endif
               .eval          count+1,count
               .if            (count==8)
               .wmsg          "WARNING!!, Out of the memory range. Use the default value 'memory_location'=0"
               .asg           "TEMP0",tmp_reg
               .break
               .endif
               .endloop
               PUSH           AR0
               LDIU           @tmp_reg,AR0
               ADDI           one,AR0
               STI            AR0,@tmp_reg
               POP            AR0
               .endm

GETMAX         .macro         memory_location, data
               .var           count, list, tmp_reg
               .asg           0,count
               .asg           "TEMP0, TEMP1, TEMP2, TEMP3, TEMP4, TEMP5, TEMP6, TEMP7",list
               .loop
               .break         ($ismember(tmp_reg,list)==0)
               .if            (memory_location==count)
               .break
               .endif
               .eval          count+1,count
               .if            (count==8)
               .wmsg          "WARNING!!, Out of the memory range. Use the default value 'memory_location'=0"
               .asg           "TEMP0",tmp_reg
               .break
               .endif
               .endloop
               CMPI           @tmp_reg,data
               BLE            lessthan?
```

```
greater?     STI          data,@tmp_reg
lessthan?
             .endm

GETMIN       .macro       memory_location, data_reg
             .var         count, list, tmp_reg
             .asg         8,count
             .asg         "TEMP8, TEMP9",list
             .loop
             .break       ($ismember(tmp_reg,list)==0)
             .if          (memory_location==count)
             .break
             .endif
             .eval        count+1,count
             .if          (count==10)
             .wmsg        "WARNING!!, Out of the memory range. Use the default value 'memory_location'=8"
             .asg         "TEMP8",tmp_reg
             .break
             .endif
             .endloop
             CMPI         @tmp_reg,data_reg
             BGE          greater?
lessthan?    STI          data_reg,@tmp_reg
greater?
             .endm

*
* Download data into 16k buffer with changing the status reg. Used for monitoring the data only.
*            DUMP(list_of_registers)
*
DUMP         .MACRO       register_list
             .var         register

             PUSH         AR1

** Where TEMP_ST and TEMP_ED are defined in the prog.

             LDI          @TEMP_ST,AR1
             CMPI         @TEMP_ED,AR1
             BZ           Delay?

             .loop
             .break       ($ismember(register, register_list) == 0)
             .if          ($isname(register)==0)
             .wmsg        "WARNING!! a non-register is in the list"
             .else
             STI          register,*AR1++
             .endif
             .endloop
             STI          AR1,@TEMP_ED
Delay?
             POP          AR1
             .endm
```

List 4: TMS320C30 DSP constant and symbol header file for the enhanced linearizer and pre-calibration DSP program.

File Name: Const.h30

```
mv_len          .set      06A0h                 ; The size of transmission to internal memory
astloc          .set      00000160h             ; Starting address for transferring the prog and data

abusctl         .set      00808060h             ; Bus control base address
primwd          .set      0800h                 ; The value for the primary bus control register
expwd           .set      0000h                 ; The value for the exapanse bus control register
primary         .set      4                     ; Primary bus control reg. address offset
expanse         .set      0                     ; Expanse bus control reg. address offset

atm0ctl         .set      00808020h             ; Timer0 base address
atm1ctl         .set      00808030h             ; Timer1 base address
tm_stop         .set      0606h                 ; Timer control reg. value to stop the timers
tm_strt         .set      06C1h                 ; Timer control reg. value to start the timers
period          .set      55                    ; 55 Timer1 period = 1/151.38899kHz => 151.38899 ksps
tm_prd          .set      8                     ; Offset Address for Timer reg

ac26            .set      00040000h             ; c26 base address
c26_dt          .set      0                     ; Data address reg. offset
c26_rst         .set      1                     ; Reset address reg. offset
c26_tgl         .set      2                     ; Toggle enable address offset

XF_cnf          .set      0006h                 ; Set the XF reg. XF0=output=1, XF1=input
dsptopc         .set      0004h                 ; XF0
pctodsp         .set      0080h                 ; XF1

admaadr         .set      00808000h             ; DMA base address
DMA_st          .set      0453h                 ; DMA control/status register value
DMA_ctl         .set      0                     ; Control address reg. offset
DMA_src         .set      4                     ; Source transfer address reg. offset
DMA_dst         .set      6                     ; Destination transfer address reg. offset
DMA_cnt         .set      8                     ; Counter1 reg. address offset

aadcadr         .set      00804000h             ; 16 bits ADC/DAC base address
dac_chA         .set      0                     ; DAC channel A data reg. offset
dac_chB         .set      1                     ; DAC channel B data reg. offset
adc_chA         .set      0                     ; ADC channel A data reg. offset
adc_chB         .set      1                     ; ADC channel B data reg. offset

ach4adr         .set      00800000h             ; 12 bits ADC/DAC base address
cch4cal         .set      00200000h             ; The values for calibrating and initialing four channel
init_ch         .set      00410000h             ;    card.
cch4int         .set      00400000h
cch4tog         .set      00010000h
ch4_ctl         .set      0                     ; Four channel card control reg. offset
ch4_sts         .set      0                     ; Four channel card status reg. offset
ch4_adc         .set      2                     ; Four channel card data reg. offset

tm0_int         .set      0100h                 ; The value for enabling the timer0 interrupt
dis_ovm         .set      0ff7fh                ; Disable the overflow mode.
en_stts         .set      2800h                 ; The value for the status reg. (ST). This value is used to
                                                ;    enable the cache and GIE
GIE             .set      2000h                 ; The value for General Interrupt Enable reg.

ablk0           .set      000008d0h             ; Address for block 0 data transfer
ablk1           .set      000000d0h             ; Address for block 1 data transfer
blksize         .set      0800h                 ; The size of each block
hlfsize         .set      0400h                 ; Half the size of each block

                .if       calib = on
nosmpwd         .set      4                     ; Numbers of sampling word save in the delay bufer
                .endif

                .if       linear = on
nosmpwd         .set      5                     ; Numbers of sampling word save in the delay bufer
                .endif

nolpsmp         .set      100                   ; The max number of allowance for the looped sample
nomxsmp         .set      62                    ; The max number of allowance for a set of sampling data
strtdly         .set      1024                  ; Numbers of delay before the sampling procedure starts
delay           .set      7                     ; The actual time slot delay values
**
** Note: This delay was generated due to the delay in the re-constructing and anti-aliasing filters.
**              From the specification, the 3dB cut-off location is at 30.91kHz (R=2kohm) for both filters.
**              The group delay of each filter below 11kHz is about 23 us. The total group delay is 46 us.
**              Since the sample frequency is 151.38899 kHz (i.e. 6.6055us period), the sampling delay of a
**              direct feedback without any distortion hooked up is 46/6.6055=7. So, This is the time slot delay
**              used in the program.
**
                .if       calib = on
dlyest          .set      2                     ; Delay estismation object used for calibrating the time delay
                                                ; Not used in the linearization program
                .endif

                .if       linear = on
dlyest          .set      0                     ; Delay estismation object used for calibrating the time delay
```

```
                                            ; Not used in the linearization program
               .endif

nospdly        .set     delay+dlyest+2      ; Numbers of the time slot delay between two samples
                                            ;    The extra two time slots are used for
                                            ;    the program simpfication and linear
                                            ;    interpolation
nostbl         .set     10                  ; Number of sampling table in the single linked table
nostwd         .set     3                   ; Number of sampling word save in the sampling table
stsize         .set     64                  ; Size of the sampling table (unit = data set)
sthfsz         .set     32                  ; Half size of the sampling table
stmask         .set     01fh                ; The mask for getting the sampling table location status bit
mapsize        .set     0100h
nxt_st         .set     nostwd*stsize       ; The offset location for the next sampling table address
sten_of        .set     nostwd*stsize+1     ; The offset location for the sampling table enable flag from
                                            ;    the top sampling table.

upper          .set     0ffffh              ; Combine with ANDN command to get the upper 16 bits data from a word
lower          .set     0ffffh              ; Combine with AND command to get the upper 16 bits data from a word

left1          .set     1                   ; Left shift 1 bit
left16         .set     16                  ; Left shift 16 bits
right16        .set     -16                 ; Rigth shift 16 bits
right20        .set     -20                 ; Right shift 20 bits; used in converting 12bits 2'c data
                                            ;    to 32 bits 2'c data
tuncat         .set     -19                 ; Tuncate 19 bit ins order for finding the look-up table index
idxof_s        .set     -21                 ; Left 21 bits in order for finding the A^2 index in sampling task
idxof_e        .set     -21                 ; Left 21 bits in order for finding the A^2 index in envelope task


               .if      calib = on
Envlp          .set     nosmpwd             ; The offset location for storing the Envelope DAC data
QIo            .set     nosmpwd-1           ; The offset location for storing the origin modulated I,Q data
QIp            .set     nosmpwd-2           ; The offset location for storing the predistorted I,Q data
QIa            .set     nosmpwd-3           ; The offset location for storing the feedback sampled DAC data
d_offset       .set     2                   ; The offset between the end of the delay buffer and the top of mapping
                                            ;    table.

Ri_2           .set     nosmpwd*5+QIa
Ri_1           .set     nosmpwd*4+QIa
Ri0            .set     nosmpwd*3+QIa
Ri1            .set     nosmpwd*2+QIa
Ri2            .set     nosmpwd*1+QIa
Ri3            .set     QIa
               .endif

               .if      linear = on
Envlp          .set     nosmpwd             ; The offset location for storing the Envelope DAC data
QIo            .set     nosmpwd             ; The offset location for storing the origin modulated I,Q data
Qa             .set     nosmpwd-1           ; The offset location for storing the feedback sampled Q data
Ia             .set     nosmpwd-2           ; The offset location for storing the feedback sampled I data
Qp             .set     nosmpwd-3           ; The offset location for storing the predistorted Q data
Ip             .set     nosmpwd-4           ; The offset location for storing the predistorted I data
d_offset       .set     5                   ; The offset between the end of the delay buffer and the top of mapping
                                            ;    table.
               .endif

zero           .set     0                   ; The symbolic value for zero
one            .set     1                   ; The symbolic value for one
n_one          .set     -1                  ; The symbolic value for negative one

cthshd2        .set     089544h             ; = 750^2 = 562500
*cthshd2       .set     03d090h            .; = 500^2 = 250000

cgIa           .set     9.78054289e-1       ; Gain value for the linear interpolation
cgQa           .set     9.89235288e-1       ; These values are found by using statistical method
cdIa           .set     2.485e-1            ; Delta value for the linear interpolation
cdQa           .set     1.975e-1            ; These values are found by using statistical method
cgEm           .set     1.311791693e1
cgEd           .set     1.31443727e1
cdEm           .set     -3.74e-1
cdEd           .set     -4.30e-1

               .if      calib = on
cdlydmp        .set     4.65661287308e-10   ; 1/(2*32768*32768)

               .asg     RS,dpadr_r          ; dual port starting address
               .asg     RS,set_en
               .asg     RS,sets2on

               .asg     RE,zero_r
               .asg     RC,bs_r

               .asg     AR0,objtmpa
               .asg     AR0,tm_adr          ; Timer0 base address

               .asg     AR1,dly_adr
               .asg     AR1,map_adr

               .asg     AR2,c26_adr         ; C26 base address reg.

               .asg     AR3,adc_adr         ; ADC & DAC base address reg.
```

```
              .asg     AR4,ch4_adr

              .asg     AR5,tmpa_2

              .asg     AR6,tmpa_1
              .asg     AR6,DMA_adr          ; DMA base address

              .asg     AR7,tmpa_0           ; Temp. reg0
              .asg     AR7,tbl_adr

              .asg     R0,objtmpr

              .asg     R1,RR1

              .asg     R2,destr_1

              .asg     R3,destr_0

              .asg     R4,tmpr_3            ; Temp. reg0
              .asg     R4,mod_Q             ; rx_I data from pi/4 DQPSK modulator

              .asg     R5,tmpr_2            ; Temp. reg1
              .asg     R5,mod_I             ; rx_Q data from pi/4 DQPSK modulator

              .asg     R6,tmpr_1            ; Temp. reg2
              .asg     R6,rx_Q

              .asg     R7,tmpr_0            ; Temp. reg3
              .asg     R7,rx_I

              .asg     IR0,idx_tbl
              .asg     IR1,IIR1

              .endif

              .if      linear = on
              .asg     RS,dpadr_r           ; dual port starting address
              .asg     RS,set_en
              .asg     RS,sets2on

              .asg     RE,zero_r
              .asg     RC,bs_r

              .asg     AR0,objtmpa
              .asg     AR0,tm_adr           ; Timer0 base address

              .asg     AR1,AAR1

              .asg     AR2,dly_adr
              .asg     AR2,map_adr

              .asg     AR3,c26_adr          ; C26 base address reg.

              .asg     AR4,adc_adr          ; ADC & DAC base address reg.

              .asg     AR5,ch4_adr

              .asg     AR6,tmpa_1
              .asg     AR6,DMA_adr          ; DMA base address

              .asg     AR7,tmpa_0           ; Temp. reg0
              .asg     AR7,tbl_adr

              .asg     R0,objtmpr

              .asg     R1,RR1

              .asg     R2,point0
              .asg     R2,tmpr_s2

              .asg     R3,tmpr_s1
              .asg     R3,point1

              .asg     R4,tmpr_3            ; Temp. reg0
              .asg     R4,mod_Q             ; rx_I data from pi/4 DQPSK modulator

              .asg     R5,tmpr_2            ; Temp. reg1
              .asg     R5,mod_I             ; rx_Q data from pi/4 DQPSK modulator

              .asg     R6,tmpr_1            ; Temp. reg2
              .asg     R6,rx_Q

              .asg     R7,tmpr_0            ; Temp. reg3
              .asg     R7,rx_I

              .asg     IR0,idx_tbl
              .asg     IR0,idx_s
              .asg     IR1,idx_e

              .endif

*
* After TM0_ISR, R4 and R5 are acted as mod_I and mod_Q;
* tmpr_1 and tmpr_0 are only the R regs used for temp reg.
*
```

List 5: TMS320C30 DSP interrupt vectors header file for the enhanced linearizer and pre-calibration DSP program.
File Name: int_vec.h30

```
*** Interrupt Vectors at address 0 ***

             .sect        ".init"

RESET        .word        Start                ; The starting address of the program sector 'Start'
                                               ;   when the RESET asserted
INT0         .word        Main
INT1         .word        TxRxdata             ; The transmitting and receiving ISR address when the
                                               ;   interrupt 1 signal asserted
INT2         .word        Main
INT3         .word        Main
XINT0        .word        Main                 ; For safty, the rest of the ISR addresses are pointing
                                               ;   to the 'Main' program section. In case, there may
RINT0        .word        Main                 ;   have some unexpected signal asserted into the
                                               ;   interrupt
XINT1        .word        Main
RINT1        .word        Main
TINT0        .word        Tmr0_isr             ; The timer 0 ISR address when the timer 0 asserts an
                                               ;   interrupt.
TINT1        .word        Main
DINT         .word        Main
```

List 6: TMS320C30 DSP memory allocation header file for the enhanced linearizer and pre-calibration DSP program.

File Name: Memory.h30

```
                .sect           ".edata"

BUS_CTL         .word           abusctl                 ; Bus control registers
ST_LOC          .word           astloc                  ; The starting address location for transfer
MAIN_AD         .word           Main                    ; The main program address location

* Dual port memory assignment

                .sect           ".dual"
PRDTBL          .space          blksize                 ; Reserve 2k word size of memory in dual port in order for table
                                                        ;  transfer.
ETEN            .space          one                     ; ??? Temperately define a memory location for the envelope
                                                        ;  table enable flag.
ENG_EN          .space          one                     ; A memory location to enable the power computation task.
NO_SYM          .space          one                     ; Numbers of symbol counted during the last enable power
                                                        ;  computation task
ENG_I0          .space          one                     ; I channel accummulated energy
ENG_I1          .space          one
ENG_Q0          .space          one                     ; Q channel accummulated energy
ENG_Q1          .space          one

                .if             calib = on
ACCS_I0         .space          one                     ; Ch I accummulated signal
ACCS_I1         .space          one
ACCS_Q0         .space          one                     ; Ch Q accummulated signal
ACCS_Q1         .space          one

DEST_EN         .space          one
DEST_I          .space          one
DEST_Q          .space          one
                .endif

                .if             linear = on
REVD            .space          (4096-blksize-nostbl*(nostwd*stsize+2)-7)         ; Reserve for the other uses.
                SLSTBL          nostbl,nostwd,stsize                              ; Generate a sampling table by
                                                                                  calling a macro function
                .endif

*
* The memory section .map and .data are first load into the external memory. After reset, these blocks of
*               memory are then transfered to the internal C30 memory in order to speed up the computation
*               speed.
*
* In .map section, there are delay buffer, mapping table and some registers
*
                .sect           ".map"
DLY_BUF         .space          (nospdly-1)*nosmpwd             ; The beginning location for the delay buffer
DLY_END         .space          nosmpwd                         ; The ending location for the delay buffer
Ao2             .space          one                             ; The memory location for A0^2

                .if             calib = on
DUMMY1          .space          one                     ; A dummy memory location
                .endif

                .if             linear = on
DUMMY4          .space          4                       ; four dummy memory locations
*
* This map is generated by using statistical method in order to equalize the  non-even
* magnitude distribution. As the result, the sampling procedure will not be pushed into
* the dead loop condition.
*
EMAP            .word           00,00,00,00,00,00,00,00
                .word           00,00,00,00,00,00,00,00
                .word           01,01,01,01,01,01,02,02
                .word           02,02,02,03,03,03,03,04
                .word           04,04,04,04,05,05,05,05
                .word           06,06,06,06,07,07,07,07
                .word           08,08,08,08,09,09,09,09
                .word           10,10,10,10,10,11,11,11
                .word           11,12,12,12,12,12,13,13
                .word           13,13,13,14,14,14,14,14

                .word           14,15,15,15,15,15,16,16
                .word           16,16,16,17,17,17,17,18
                .word           18,18,19,19,19,19,20,20
                .word           20,21,21,22,22,23,23,23
                .word           24,24,25,25,26,27,27,28
                .word           28,29,29,30,30,31,31,32
                .word           33,33,33,34,34,35,35,36
                .word           36,37,37,38,38,38,39,39
                .word           39,40,40,41,41,41,42,42
                .word           42,43,43,43,43,44,44,45

                .word           45,45,46,46,46,47,47,47
                .word           48,48,48,49,49,50,50,50
                .word       51,51,51,52,52,52,53,53
                .word       54,54,54,55,55,55,56,56
                .word       56,57,57,57,58,58,58,58
                .word       59,59,59,59,60,60,60,60
                .word       60,61,61,61,61,61,61,62
                .word       62,62,62,62,62,62,62,63
                .word       63,63,63,63,63,63,63,63
                .word       63,63,63,63,63,63,63,63

                .word       63,63,63,63,63,63,63,63
                .word       63,63,63,63,63,63,63,63
                .endif

*
* In .data section, it stores address locations, coefficients, flags, object and task pointers, temporary memory for
debug and
*               system stack.
*
                .data
G_Ia            .float      cgIa                ; Gain value for the linear interpolation
G_Qa            .float      cgQa
G_Em            .float      cgEm
G_Ed            .float      cgEd
D_Ia            .float      cdIa                ; Delta value for the linear interpolation
D_Qa            .float      cdQa
D_Em            .float      cdEm
D_Ed            .float      cdEd

                .if         calib = on
Dlydmp          .float      cdlydmp
N_ONE           .word       0ffffffffh
                .endif

                .word       0ddddddddh          ; For seperation only while using the mon30 to debug the prog.
DUAL_PT         .word       PRDTBL              ; Predistorted table starting address
CH4ADR          .word       ach4adr             ; The base address for 4 channel board
TM0_CTL         .word       atm0ctl             ; Timer0 control register
TM1_CTL         .word       atm1ctl             ; Timer1 control register
DMAADR          .word       admaadr             ; DMA base address
ADCADR          .word       aadcadr             ; ADC and DAC base address
C26             .word       ac26                ; C26 base address
DLY_ST          .word       DLY_BUF             ; The 1st address location in the delay buffer
DLY_STP         .word       DLY_END             ; The last address location in the delay buffer
ETENADR         .word       ETEN                ; The address of envelope table enable flag
EGENADR         .word       ENG_EN              ; The address of power computation task enable flag

                .if         calib = on
EST_IADR        .word       DEST_I
SQ1DADR         .word       ACCS_Q1
SQ1ADR          .word       ACC_Q1
                .endif

                .if         linear = on
SMPTADR         .word       STBL0               ; The base address location for sampling table
                .endif

RSP             .word       STACK               ; Stack address pointer

                .word       0fffffffh           ; For seperation only while using the mon30 to debug the prog.
CH4_CAL         .word       cch4cal             ; Toggle the calibration bit
CH4_AD0         .word       init_ch             ; Select ADC0
CH4_INT         .word       cch4int             ; The initial condition for the four channels card
CH4_TOG         .word       cch4tog             ; Toggle the next sampling channel
CH_SEL          .word       init_ch             ; Select which channel in 4 channel card will be used

THSHLD2         .word       cthshd2             ; The threshold value for sampling data
SMP_CNT         .word       zero                ; Sampling counter
LOOP_S          .word       zero                ; Looped sampling counter
SMP_EN          .word       -strtdly            ; Read Enable for 30 pts data table
                                                ; If SMP_EN < 0 not sampling,
                                                ; if SMP_EN == 0 read data from DAC,
                                                ; if SMP_EN > 1 save the sampling data,
                                                ; -32 Used to delay when the feedback sampling start
ENV_CNT         .word       zero                ; envelope status counter
OBJ2_ON         .word       zero                ; End of object2 task flag
SMP_STS0        .word       zero                ; The sampling status memory for the sampling data set
SMP_STS1        .word       zero                ; The sampling status memory for the sampling data set

ENG_FLG         .word       zero                ; The flag for enable the energy computational routine in the ISR
                                                ; 1 for enable; 0 for disable.
OUT_CNT         .word       zero                ; The numbers of symbol counted
ACCE_I0         .word       zero                ; The accummuate I channel energy
ACCE_I1         .word       zero
ACCE_Q0         .word       zero                ; The accummuate Q channel energy
ACCE_Q1         .word       zero

                .if         calib = on
ACC_I0          .word       zero
ACC_I1          .word       zero
ACC_Q0          .word       zero
ACC_Q1          .word       zero
                .endif
```

```
            .word       0cccccccch      ; For seperation only while using the mon30 to debug the prog.
OBJ0PTR     .word       Comm0           ; object0 task pointer
COMM0       .word       Comm0           ; Intermidiate task location for COMM task
COMM1       .word       Comm1
COMM2       .word       Comm2
COMM3       .word       Comm3
COMM4       .word       Comm4

            .word       01111111h       ; For seperation only while using the mon30 to debug the prog.
OBJ1PTR     .word       Energy0         ; object1 task pointer
ENERGY0     .word       Energy0
ENERGY1     .word       Energy1
ENERGY2     .word       Energy2
ENERGY3     .word       Energy3
ENERGY4     .word       Energy4

            .if         calib = on
            .word       02222222h       ; For seperation only while using the mon30 to debug the prog.
TMP_S1      .word       zero            ; Temporary storage memory for object 1
OBJ2PTR     .word       DlyEst0         ; object1 task pointer
DLYEST0     .word       DlyEst0
DLYEST1     .word       DlyEst1
DLYEST2     .word       DlyEst2
DLYEST3     .word       DlyEst3
DLYEST4     .word       DlyEst4
DLYEST5     .word       DlyEst5
DLYEST6     .word       DlyEst6
DLYEST7     .word       DlyEst7
DLYEST8     .word       DlyEst8
DLYEST9     .word       DlyEst9
            .endif

            .if         linear = on
            .word       02222222h       ; For seperation only while using the mon30 to debug the prog.
TMP_S1      .word       zero            ; Temporary storage memory for object 1
OBJ2PTR     .word       Check0          ; object1 task pointer
CHECK0      .word       Check0          ; Intermidiate task location for CHECK task
CHECK1      .word       Check1
CHECK2      .word       Check2
SMPL0       .word       Smpl0           ; Intermidiate task location for SAMPLING task
SMPL1       .word       Smpl1           ; Intermidiate task location for SAMPLING task
SMPL2       .word       Smpl2
SMPL3       .word       Smpl3
CALC_S0     .word       Calc_s0         ; Intermidiate task location for CALC_S task
CALC_S1     .word       Calc_s1
SMPL4       .word       Smpl4
SMPL5       .word       Smpl5
SMPL6       .word       Smpl6
SMPL7       .word       Smpl7
WAIT        .word       Wait            ; WAIT task
RE_EN       .word       Re_en           ; RE_ENable task pointer

            .word       03333333h       ; For seperation only while using the mon30 to debug the prog.
TMP_S2      .word       zero            ; Temporary storage memory for object 2
OBJ3PTR     .word       Env0            ; object2 task pointer
ENV0        .word       Env0            ; Intermidiate task location for ENVELOPE task
CALC_E0     .word       Calc_e0         ; Intermidiate task location for CALC_S task
CALC_E1     .word       Calc_e1
CALC_E2     .word       Calc_e2
ENV1        .word       Env1            ; Intermidiate task location for ENVELOPE task
ENV2        .word       Env2
ENV3        .word       Env3
            .endif

            .word       0f0f0f0f0f0h    ; For seperation only while using the mon30 to debug the prog.
TEMP_ST     .word       00004000h       ; The starting address for the 16k buffer
TEMP_ED     .word       00008000h       ; The ending address for the 16k buffer
TEMP0       .word       zero            ; Temporary storage for general use
TEMP1       .word       zero
TEMP2       .word       zero
TEMP3       .word       zero
TEMP4       .word       zero
TEMP5       .word       zero
TEMP6       .word       zero
TEMP7       .word       zero
TEMP8       .word       07fffffffh
TEMP9       .word       07fffffffh
            .word       0f0f0f0f0f0h    ; For seperation only while using the mon30 to debug
                                        ; the prog.
STACK       .word       zero            ; The beginning of the stack
```

List 7: TMS320C30 DSP initialization file for the enhanced linearizer and pre-calibration DSP program.
File Name: Initial.c30

```
* External Program Section at address 0D0h
*
                .sect         ".etext"
Start:
*
* Set up DP
*
                LDP           BUS_CTL                ; Set up page pointer for direct addressing
*
* Set up address bus configuation
*
                LDI           @BUS_CTL,tmpa_0        ; Set up primary and expansion bus
                LDI           primwd,tmpr_0          ; wait states.
                LDI           expwd,tmpr_1           ; Since expwd == 0, tmpr_1 == 0
                STI           tmpr_0,*+tmpa_0(primary)
                STI           tmpr_1,*+tmpa_0(expanse)
*
* Transfer the linearization program and required parameters to the internal
* memory in order to speed up the process. Since the "CoffLoad" occupies the
* internal while downloading the program into the memory, this transfer is
* necessary.
*
                LDI           @ST_LOC,tmpa_0         ; Transfer the program and data into the
                LDI           @MAIN_AD,tmpa_1        ;       internal memory
                LDI           *tmpa_0++,tmpr_0
                RPTS          mv_len-1
                LDI           *tmpa_0++,tmpr_0
||              STI           tmpr_0,*tmpa_1++

                LDI           @MAIN_AD,tmpa_0        ; Start the main routine
                B             tmpa_0
*
* Internal Program Section at address 0809800h
*
                .text
*
* For the initialization, the pipeline conflict is less to be concerned.
*
Main:
*
*Set up DP and SP
*
                LDP           DUAL_PT                ; Set up page pointer for direct addressing
                LDI           @RSP,SP                ; Set up stack pointer.
*
* Reset C26 modulator
*
                LDI           @C26,c26_adr           ; Reset the c26 board
                STI           tmpr_0,*+c26_adr(c26_rst)

                LDI           @CH4ADR,ch4_adr        ; Toggle the calibration bit in the control reg.
                LDI           @CH4_CAL,tmpr_0        ;       to calibrate the 4 channel board
                LDI           @CH4_AD0,tmpr_2
                STI           tmpr_1,*+ch4_adr(ch4_ctl)
                STI           tmpr_0,*+ch4_adr(ch4_ctl)
                STI           tmpr_1,*+ch4_adr(ch4_ctl)

WAITCAL:
                TSTB          *+ch4_adr(ch4_sts),tmpr_0     ; Wait for finishing calibration
                BZ            WAITCAL
                STI           tmpr_2,*+ch4_adr(ch4_ctl)     ; Enable channel 0 for the 1st sampling channel
*
* The following program section is used to initialize the initial requirments for the
* linearization routine. Besides, this program section is also used to wait for being the
* C26 stable. This minimum time is approximated to 1.8us (i.e. 30 instruction-cycles of
* C30-33).
*
* Set up DAC channel A,B
*
                LDI           @ADCADR,adc_adr                   ; Set ponter to DAC
                LDI           0,tmpr_0                          ; Set initial value
                STI           tmpr_0,*+adc_adr(dac_chA)         ; Out Chan A,B
||              STI           tmpr_0,*+adc_adr(dac_chB)
                LDI           tmpr_0,ST                         ; initial the cache status
*
* Set up XF0 (as output=1) and XF1 (as input)
*
                LDI           XF_cnf,IOF             ; Initial the PC link control signal
*
* Set up the initial DMA condition.
*
                LDI           @DMAADR,DMA_adr                   ; Load DMA base address
                LDI           @DUAL_PT,tmpr_0                   ; Load the source and destination
                                                                ; addresses
                LDI           ablk1,tmpr_1
                STI         tmpr_0,*+DMA_adr(DMA_src)       ; Store the source and destination
                                                            ; addresses
                STI         tmpr_1,*+DMA_adr(DMA_dst)
                LDI         zero,tmpr_0                     ; Load the DMA operation condition values
                LDI         DMA_st,tmpr_1
                STI         tmpr_0,*+DMA_adr(DMA_cnt)       ; Set the DMA operation condition
                STI         tmpr_1,*+DMA_adr(DMA_ctl)
*
* Set up timer0 and timer1
*
                LDI         @TM0_CTL,tm_adr             ; Load Timer0 address
                LDI         tm_stop,tmpr_0              ; Load the start and stop condition
                LDI         tm_strt,tmpr_1
                LDI         @TM1_CTL,tmpa_0             ; Load Timer1 address
                LDI         period*29,tmpr_2            ; Load the period values
                LDI         period,tmpr_3
                LDI         tm_prd,IR0                  ; Load Index reg 0
                STI         tmpr_0,*tm_adr              ; Stop both timers
||              STI         tmpr_0,*tmpa_0
                STI         tmpr_2,*+tm_adr(IR0)        ; Set the period for the timers
||              STI         tmpr_3,*+tmpa_0(IR0)
                STI         tmpr_1,*tm_adr              ; Start both timers
||              STI         tmpr_1,*tmpa_0

*
* DON'T CHANGE tmpr_0 VALUE BEFORE TM0_ISR is performed
*               ( since tmpr_0 holds the tm_stp value )
*
*
* Set the required parameters into the appropriated registers.
*
* Assume the initial table (i.e. the first calibrated table is located at BLK0, and
* the DMA destination address is pointing to BLK1 location.
*
                LDI         ablk0,tbl_adr           ; Set the next DMA transfer destination addr.
                LDI         @DUAL_PT,dpadr_r        ; Set the dual port addr. into dp register.
                LDI         blksize,bs_r            ; Set the block size value into bs register.
                                                    ;       into pdn addr register.
                LDI         @DLY_ST,dly_adr         ; Set the delay table address pointer
*
* Set up interrupt condition
*
                LDI         zero,IF                 ; CLear all pending interrupt
                LDI         tm0_int,IE              ; Enable interrupt for timer0
*
* Enable the modulator
*
                LDI         zero,mod_I                          ; Initial the output I and Q signals
                LDI         zero,mod_Q
                LDI         zero,zero_r                         ; Set the zero reg.
                STI         tmpr_0,*+c26_adr(c26_tgl)           ; Enable c26 (i.e. start the modulator)
                IDLE                                            ; Set global interrupt enable and wait
                                                                ;   for interrupts
*
* This ISR is used to eliminate the first 29 wait states that is due to the C26 latching the input serial data.
*               that's why the period of timer0 is equal to 29 times of the sampling period.
*
Tmr0_isr:
                STI         tmpr_0,*tm_adr          ; Stop TIMER0
                XOR         0102h,IE                ; Disable timer0 and enable INT1 interrupts
                LDI         zero,IF                 ; Clear all pending interrupt
                POP         tmpr_0                  ; Clear the stack
                AND         dis_ovm,ST              ; Disable OVM
                OR          en_stts,ST              ; Enable cache and GIE
```

List 8: TMS320C30 DSP pre-calibration program.

File Name: 30c.c30

```
on              .set          1
off             .set          0
linear          .set          off
calib           .set          on                     ; define this program for calibration

                .include      "macro.fcn"            ; Load the macro function from file "macro.fcn"

                .include      "const.h30"            ; load the constant assignment heading "const.h30"
                .include      "int_vec.h30"          ; Load the interrupt vector assignment table
                                                     ; "int_vec.h30"
                .include      "memory.h30"           ; Load the memory assignment heading "memory.h30"

                .include      "initial.c30"          ; Load the initialization routine "initial.c30"

                .text
*
* dsptopc == 'high' => DMA is free; dsptopc == 'low' => DMA is busy
* pctodsp == 'high' => new data; pctodsp == 'low' acknowledge C30 signal
*
Object00:
Comm0:
*
* Check whether the PC has transfer a new table into the dual port memory
*
                TSTB          pctodsp,IOF                    ; Check whether PC has signaled to DSP
                LDIU          @OBJ1PTR,objtmpr               ; Switch to object1 task
                BD            objtmpr
                LDIZ          @COMM0,objtmpa                 ; Without new table, next go back to
                                                             ; object0::Comm0
                LDINZ         @COMM1,objtmpa                 ; With new table, next go to object::Comm1
                STI           objtmpa,@OBJ0PTR               ; Update object0 pointer

Object01:
Comm1:
*
* A new table arrived, start the DMA transfer and acknowledge to PC
*
                STI           bs_r,*+DMA_adr(DMA_cnt)            ; Enable the DMA
                ANDN          dsptopc,IOF                        ; Assert a 'low' into XF0

Object02:
Comm2:
*
* Check whether the PC acknowledge the DSP signal
*
                TSTB          pctodsp,IOF            ; ? XF1 == 'low'
                LDIU          @OBJ1PTR,objtmpr       ; Switch to object1 task
                BD            objtmpr
                LDINZ         @COMM2,objtmpa         ; Without Ack from PC, next go back to
                                                     ; object0::Comm2
                LDIZ          @COMM3,objtmpa         ; With Ack from PC, next go to object0::Comm3
                STI           objtmpa,@OBJ0PTR       ; Update object0 pointer

Object03:
Comm3:
*
* Check whether the DMA has finished its transfer
*
                CMPI          *+DMA_adr(DMA_cnt),zero_r           ; Transfer finish ?
                LDIU          @OBJ1PTR,objtmpr                    ; not finished, Switch to object1 task
                BD            objtmpr
                LDINZ         @COMM3,objtmpa                      ; Not finish transfer, next go back to
                                                                  ; object0::Comm3
                LDIZ          @COMM4,objtmpa                      ; Finish transfer, next go to
                                                                  ; object0::Comm4
                STI           objtmpa,@OBJ0PTR                    ; Update object0 pointer

Object04:
Comm4:
*
* Send a signal to PC to indicate the transfer is completed and dual port memory is free
*
** Note avoid reg. conflict
                STI           dpadr_r,*+DMA_adr(DMA_src)          ; Set the source dual port address
                OR            dsptopc,IOF                         ; Assert a 'high' into XF0
                LDIU          @ENERGY1,objtmpr                    ; Switch to object1::Energy1
                STI           tbl_adr,*+DMA_adr(DMA_dst)          ; Set the destination table address
                BD            objtmpr
                LDIU          @COMM0,objtmpr                      ; Next go to object0::Comm0
                STI           objtmpr,@OBJ0PTR                    ; Update object0 pointer
                XOR           blksize,tbl_adr                     ; Update table address pointer

Object10:
Energy0:
                CMPI          @SMP_EN,zero_r         ; Check whether system is in stable condition
                LDIU          @OBJ2PTR,objtmpr       ; Switch to object2
                BD            objtmpr
                LDIGT         @ENERGY0,objtmpr        ; If not stable, next go to object1::Energy0
                LDILE         @ENERGY1,objtmpr        ; If stable, next go to object1::Energy1
                STI           objtmpr,@OBJ1PTR        ; Update the object1 pointer

Object11:
Energy1:

                STI           zero_r,@OUT_CNT         ; Set the symbol counter (OUT_CNT) to zero
                STI           zero_r,@ACCE_I0         ; Set the accumulate energy I variable (ACCE_I) to
                                                      ; zero
                STI           zero_r,@ACCE_I1
                STI           zero_r,@ACCE_Q0         ; Set the accumulate energy Q variable (ACCE_Q) to
                                                      ; zero
                STI           zero_r,@ACCE_Q1
                STI           zero_r,@ACC_I0          ; Set the accumulate signal I variable (ACC_I) to
                                                      ; zero
                STI           zero_r,@ACC_I1
                STI           zero_r,@ACC_Q0          ; Set the accumulate signal Q variable (ACC_Q) to
                                                      ; zero
                STI           zero_r,@ACC_Q1
                LDIU          one,objtmpa             ; Load one into objtmpa which is used to enable
                                                      ; the energy computational routine in ISR
                LDIU          @OBJ2PTR,objtmpr        ; Switch to object2 task
                BD            objtmpr
                STI           objtmpa,@ENG_FLG        ; Enable the energy and signal computation routine
                                                      ; in ISR
                LDIU          @ENERGY2,objtmpr        ; Next go to object1::Energy2
                STI           objtmpr,@OBJ1PTR        ; Update the object1 pointer

Object12:
Energy2:
                LDIU          @EGENADR,objtmpa
                LDIU          @OBJ2PTR,objtmpr
                CMPI          *objtmpa,zero_r
                BD            objtmpr
                LDIZ          @ENERGY3,objtmpr        ; If disable, next go to object1::Energy3
                LDINZ         @ENERGY2,objtmpr        ; If no request, next go the same object task
                STI           objtmpr,@OBJ1PTR        ; Update the object1 pointer

Object13:
Energy3:

                LDIU          @SQ1ADR,objtmpa         ; Load the Power enable flag address
                LDIU          @SQ1DADR,tmpa_2
                STI           zero_r,@ENG_FLG         ; Disable the energy comptation task in ISR

; Transfer the accumulated signals of channel I and Q, the accumulated energy of channel I,Q and number of symbol
counted to the dual port
;               memory location for 486 access

                LDI           *objtmpa--,objtmpr      ; Load ACCS_Q1
                LDI           *objtmpa--,objtmpr      ; Load ACCS_Q0
||              STI           objtmpr,*tmpa_2--       ; Save ACCS_Q1
                LDI           *objtmpa--,objtmpr      ; Load ACCS_I1
||              STI           objtmpr,*tmpa_2--       ; Save ACCS_Q0
                LDI           *objtmpa--,objtmpr      ; Load ACCS_I0
||              STI           objtmpr,*tmpa_2--       ; Save ACCS_I1
                LDI           *objtmpa--,objtmpr      ; Load ACCE_Q1
||              STI           objtmpr,*tmpa_2--       ; Save ACCS_I0
                LDI           *objtmpa--,objtmpr      ; Load ACCE_Q0
||              STI           objtmpr,*tmpa_2--       ; Save ACCE_Q1
                LDI           *objtmpa--,objtmpr      ; Load ACCE_I1
||              STI           objtmpr,*tmpa_2--       ; Save ACCE_Q0
                LDI           *objtmpa--,objtmpr      ; Load ACCE_I0
||              STI           objtmpr,*tmpa_2--       ; Save ACCE_I1
                LDI           *objtmpa--,objtmpr      ; Load OUT_CNT
||              STI           objtmpr,*tmpa_2--       ; Save ACCE_I0
                STI           objtmpr,*tmpa_2--       ; Save OUT_CNT

                LDIU          @OBJ2PTR,objtmpr        ; Switch to object2 task
                BD            objtmpr
                STI           set_en,*tmpa_2          ; Acknowledge to 486 that a new set of data is
                                                      ; ready
                LDIU          @ENERGY4,objtmpr        ; Next go to object1::Energy4
                STI           objtmpr,@OBJ1PTR        ; Update the object1 pointer

Object14:
Energy4:
                LDIU          @OBJ2PTR, objtmpr
                BD            objtmpr
                LDIU          @ENERGY4,objtmpr        ; Next go to object1::Energy4
                STI           objtmpr,@OBJ1PTR        ; Update the object1 pointer
                NOP

Object20:
DlyEst0:
                CMPI          @SMP_EN,zero_r          ; if sampling is not finished then branch
                LDIU          @OBJ0PTR,objtmpr        ; Switch to object0 task
                BD            objtmpr
                LDIGE         @DLYEST0,objtmpa        ; Not avaible, next go back to object2::DlyEst0
                LDILT         @DLYEST1,objtmpa        ; Avaible, next go to object2::DlyEst1
                STI           objtmpa,@OBJ2PTR        ; Update object2 pointer
```

```
Object21:
DlyEst1:
*
* The following two objects (object23::DlyEst3 and object24::DlyEst4) are used to find out
* the time delay
*               term for the channel Q iteration.
*
                LDIU        *-dly_adr(Ri_2),destr_0
                ASH         right16,destr_0                 ; term1=Ri_2

                LDIU        *-dly_adr(Ri_1),destr_1
                ASH         right16,destr_1                 ; term2=Ri_1
                SUBI        destr_1,destr_0
                SUBI        destr_1,destr_0                 ; term1=Ri_2-2*Ri_1

                LDIU        *-dly_adr(Ri1),objtmpr
                ASH         right16,objtmpr                 ; Load Ri1
                ADDI        objtmpr,destr_0
                ADDI        objtmpr,destr_0                 ; term1=Ri_2-2*Ri_1+2*Ri1

                LDIU        *-dly_adr(Ri2),objtmpr
                ASH         right16,objtmpr                 ; Load Ri2
                ADDI        objtmpr,destr_1
                ADDI        objtmpr,destr_1                 ; term2=Ri_1+2*Ri2
                LDIU        @OBJ0PTR,objtmpa
                BD          objtmpa
                SUBI        objtmpr,destr_0                 ; term1=Ri_2-2*Ri_1+2*Ri1-Ri2
                LDIU        @DLYEST2,objtmpr
                STI         objtmpr,@OBJ2PTR

Object22:
DlyEst2:
                LDIU        *-dly_adr(Ri0),objtmpr
                ASH         right16,objtmpr             ; Load Ri
                SUBI        objtmpr,destr_1
                SUBI        objtmpr,destr_1             ; term2=Ri_1-2*Ri+2*Ri2

                LDIU        *-dly_adr(Ri3),objtmpr
                ASH         right16,objtmpr             ; Load Ri3
                SUBI        objtmpr,destr_1             ; term2=Ri_1-2*Ri+2*Ri2-Ri3

                LDIU        @OBJ0PTR,objtmpa
                BD          objtmpa
                SUBI        destr_0,destr_1             ; term2=-Ri_2+3*Ri_1-2*Ri-2*Ri1+3*Ri2-Ri3
                LDIU        @DLYEST3,objtmpr
                STI         objtmpr,@OBJ2PTR

Object23:
DlyEst3:
                FLOAT       destr_0
                FLOAT       destr_1
                MPYF        @D_Qa,destr_1
                ADDF        destr_0,destr_1             ; The last term
                LDIU        *-dly_adr(Ri1),objtmpr
                ASH         right16,objtmpr             ; Load Ri1
                LDIU        *-dly_adr(Ri0),destr_0
                ASH         right16,destr_0             ; Load Ri
                SUBI        destr_0,objtmpr             ; Ri1-Ri
                FLOAT       objtmpr
                FLOAT       destr_0
                MPYF        @D_Qa,objtmpr
                ADDF        objtmpr,destr_0
                LDIU        @OBJ0PTR,objtmpa
                BD          objtmpa
                MPYF        @G_Qa,destr_0
                LDIU        @DLYEST4,objtmpr
                STI         objtmpr,@OBJ2PTR

Object24:
DlyEst4:
                LDIU        *-dly_adr(nosmpwd*(nospdly-2)+QIp),objtmpr
                ASH         right16,objtmpr         ; Load Si
                FLOAT       objtmpr
                SUBF        objtmpr,destr_0
                MPYF        destr_1,destr_0
                MPYF        @Dlydmp,destr_0
                SUBRF       @D_Qa,destr_0
                LDIU        @OBJ0PTR,objtmpa
                BD          objtmpa
                STF         destr_0,@D_Qa
                LDIU        @DLYEST5,objtmpr
                STI         objtmpr,@OBJ2PTR

Object25:
DlyEst5:
*
* The following two objects (object23::DlyEst3 and object24::DlyEst4) are used to find out
* the time delay
*               term for the channel I iteration.
*
                LDIU        *-dly_adr(Ri_2),destr_0
                LSH         left16,destr_0
                ASH         right16,destr_0             ; term1=Ri_2
                LDIU        *-dly_adr(Ri_1),destr_1
                LSH         left16,destr_1
                ASH         right16,destr_1         ; term2=Ri_1
                SUBI        destr_1,destr_0
                SUBI        destr_1,destr_0         ; term1=Ri_2-2*Ri_1

                LDIU        *-dly_adr(Ri1),objtmpr
                LSH         left16,objtmpr
                ASH         right16,objtmpr         ; Load Ri1
                ADDI        objtmpr,destr_0
                ADDI        objtmpr,destr_0         ; term1=Ri_2-2*Ri_1+2*Ri1

                LDIU        *-dly_adr(Ri2),objtmpr
                LSH         left16,objtmpr
                ASH         right16.objtmpr         ; Load Ri2
                ADDI        objtmpr,destr_1
                ADDI        objtmpr,destr_1         ; term2=Ri_1+2*Ri2
                LDIU        @OBJ0PTR,objtmpa
                BD          objtmpa
                SUBI        objtmpr,destr_0         ; term1=Ri_2-2*Ri_1+2*Ri1-Ri2
                LDIU        @DLYEST6,objtmpr
                STI         objtmpr,@OBJ2PTR

Object26:
DlyEst6:
                LDIU        *-dly_adr(Ri0),objtmpr
                LSH         left16,objtmpr
                ASH         right16,objtmpr         ; Load Ri
                SUBI        objtmpr,destr_1
                SUBI        objtmpr,destr_1         ; term2=Ri_1-2*Ri+2*Ri2

                LDIU        *-dly_adr(Ri3),objtmpr
                LSH         left16,objtmpr
                ASH         right16,objtmpr         ; Load Ri3
                SUBI        objtmpr,destr_1         ; term2=Ri_1-2*Ri+2*Ri2-Ri3

                LDIU        @OBJ0PTR,objtmpa
                BD          objtmpa
                SUBI        destr_0,destr_1         ; term2=-Ri_2+3*Ri_1-2*Ri-2*Ri1+3+Ri2-Ri3
                LDIU        @DLYEST7,objtmpr
                STI         objtmpr,@OBJ2PTR

Object27:
DlyEst7:
                FLOAT       destr_0
                FLOAT       destr_1
                MPYF        @D_Ia,destr_1
                ADDF        destr_0,destr_1             ; The last term
                LDIU        *-dly_adr(Ri1),objtmpr
                LSH         left16,objtmpr
                ASH         right16,objtmpr             ; Load Ri1
                LDIU        *-dly_adr(Ri0),destr_0
                LSH         left16,destr_0
                ASH         right16,destr_0             ; Load Ri0
                SUBI        destr_0,objtmpr             ; Ri1-Ri
                FLOAT       objtmpr
                FLOAT       destr_0
                MPYF        @D_Ia,objtmpr
                ADDF        objtmpr,destr_0
                LDIU        @OBJ0PTR,objtmpa
                BD          objtmpa
                MPYF        @G_Ia,destr_0
                LDIU        @DLYEST8,objtmpr
                STI         objtmpr,@OBJ2PTR

Object28:
DlyEst8:
                LDIU        *-dly_adr(nosmpwd*(nospdly-2)+QIp),objtmpr
                LSH         left16,objtmpr
                ASH         right16,objtmpr       ; Load Si
                FLOAT       objtmpr
                SUBF        objtmpr,destr_0
                MPYF        @Dlydmp,destr_0
                SUBRF       @D_Ia,destr_0
                LDIU        @OBJ0PTR,objtmpa
                BD          objtmpa
                STF         destr_0,@D_Ia
                LDIU        @DLYEST9,objtmpr
                STI         objtmpr,@OBJ2PTR

Object29:
DlyEst9:
                LDIU        @EST_IADR,objtmpa
                CMPI        *-objtmpa(1),zero_r        ; Check 486 request to stop ?
                BEQD        Re_en                      ; If not, go to re enable routine.
                LDIU        @OBJ0PTR,objtmpr           ; Switch to Object0
                LDIU        @DLY_ST,dly_adr            ; Reset dly_adr point to delay table
                LDFU        @D_Ia,destr_1              ; Load the most update time delay constants of
                LDFU        @D_Qa,destr_0              ; both channel I and Q.
                STF         destr_1,*objtmpa           ; Update the dual port memory locations
||              STF         destr_0,*+objtmpa(1)       ;  for 486 access
                STI         zero_r,*-objtmpa(1)        ; Acknowledge to 486 that a new time delay
```

```
                                                        ; constant is avaible.
*
* Re-enable the sampling process
*

Re_en:
                BD          objtmpr
                LDIU        @DLYEST0,objtmpr           ; Next go to Object2::DlyEst0
                STI         objtmpr,@OBJ2PTR           ; Update the object2 pointer
                STI         zero_r,@SMP_EN             ; Enable sampling routine again

*
* Interrupt Service Routine for getting data from
* pi/4 DQPSK modulator and transmitting the predistoted
* I & Q signal to the DAC
*
                .align                              ; To align the following code into cache

TxRxdata:
                PUSH        ST                      ; Save status reg. into the stack
                PUSH        idx_tbl                 ; Save idx_tbl reg. into the stack

                LDI         *+adc_adr(adc_chA),rx_I       ; In Chan A,B
||              LDI         *+adc_adr(adc_chB),rx_Q       ; Note: There is a memory conflict.
                STI         mod_I,*+adc_adr(dac_chA)      ; Out Chan A,B
||              STI         mod_Q,*+adc_adr(dac_chB)      ; Note: There is a memory conflict.

*
* The above two instructions must be placed as early as the beginning the ISR, the MAX
* seperation of these instructions from the top of ISR must be less than 20 instruction
* cycles. Note: there are above 8 cycles for acknowledging the interrupt and two wait
* state for accessing the ADC or DAC devices. As the result, there are only about 3 to 4
* single cycle instructions can be performed.
*
                LDI         *c26_adr,idx_tbl        ; load the new modulated I and Q signals
*
* This section is used to compress the two 16bits integer data into 32bits data.
* The usage of this compression is to release more registers for other use and to increase
* the data transfer rate
*               in either the delay buffer or the dual port circular buffer.
*
                ANDN        upper,rx_Q              ; Get the rx Q channel data and place into upper
                                                    ; 16bits
                ANDN        upper,mod_Q             ; Get the tx Q channel data and place into upper
                                                    ; 16bits
                LSH         right16,rx_I            ; Get the rx I channel data and place into lower
                                                    ; 16bits
                LSH         right16,mod_I           ; Get the tx I channel data and place into lower
                                                    ; 16bits
                OR          rx_Q,rx_I,rx_Q          ; Combine the rx data into one 32 bits data
                OR          mod_Q,mod_I,rx_I        ; Combine the tx data into one 32 bits data
*
* The registers mod_I and mod_Q (i.e. tmpr_2 and tmpr_3 ) are free for getting data from
* another
*               ADC devices which digitize the feedback signal from the envelope detector.
*
*               LDI         *+ch4_adr(ch4_adc),tmpr_3           ; Sampling the envelope detector
*               LDI         @CH_SEL,tmpr_2                      ; Load the next sampling channel enable
                                                                ; flag
*               STI         tmpr_2,*+ch4_adr(ch4_ctl)           ; Set the next sampling channel
*
*               DUMP        "rx_I, rx_Q, tmpr_3"                ; Download data into 16k buffer with
                                                                ; changing the status reg.

*
* This section is used to push the sampled data into the delay buffer when it is enable.
*
                STI         idx_tbl,*+dly_adr(1)         ; Backup the new modulated signal
                LDI         one,tmpr_2                   ; Set tmpr_2 to one
                CMPI        @SMP_EN,zero_r               ; Check the sampling enable condition
                BNZD        engcal                       ; if not ready to sample then jump to
                                                         ; predistion routine
                LDILE       zero,tmpr_2                  ; if SMP_EN >= 0 then without increase SMP_EN
                ADDI        @SMP_EN,tmpr_2
                STI         tmpr_2,@SMP_EN               ; Update SMP_EN value
                CMPI        @DLY_STP,dly_adr             ; Check whether the delay buffer is completely
                                                         ; fill
                BLTD        engcal                       ; if not reach the bottom then jump to 'Next'
                STI         tmpr_3,*dly_adr++(2)         ; Save the current sampling data into the
                STI         rx_I,*dly_adr++              ;  delay buffer
                STI         rx_Q,*dly_adr++
                STI         dpadr_r,@SMP_EN              ; if reach the bottom then disable sampling
*
* At this point, rx_I, rx_Q, mod_I and mod_Q (i.e. tmpr_0,tmpr_1,tmpr_2 and tmpr_3) are
* free.
*
*
* The section is used to compute the energy of the sampled I and Q channel signals.
* (Note: three commands inside this section are used for pre-calculate the LUT index
* rather than
*               compute the energy.)
*
engcal:
```

```
                LDI         @ENG_FLG,tmpr_2         ; Check whether the power computation routine is enable?
                                                    ; If ENG_FLG=1, calculate the power
                BZD         unitout                 ; If ENG_FLG=0, not calculate the power
*
* The following three commands is used to pre-calculate the LUT index.
*

                LDIU        rx_Q,rx_I               ;Duplicate the rx signal into both rx_I and rx_Q
                LSH         left16,rx_I             ; Get rx ch I signal.
                ADDI        @OUT_CNT,tmpr_2         ; Increment the symbol count variable by 1.

                STI         tmpr_2,@OUT_CNT         ; Save the updated symbol count variable.

                ASH         right16,rx_I
                LDIN        n_one,mod_I
                LDINN       zero_r,mod_I
                MPYI        rx_I,rx_I,tmpr_3        ; Calculate the I channel energy
                ADDI        @ACC_I0,rx_I
                ADDC        @ACC_I1,mod_I
                STI         rx_I,@ACC_I0
                STI         mod_I,@ACC_I1
                ADDI        @ACCE_I0,tmpr_3         ; Accummulate the I channel energy
                LDIU        @ACCE_I1,rx_I
                ADDC        zero_r,rx_I
                STI         tmpr_3,@ACCE_I0
                STI         rx_I,@ACCE_I1

                ASH         right16,rx_Q
                LDIN        n_one,mod_Q
                LDINN       zero_r,mod_Q
                MPYI        rx_Q,rx_Q,tmpr_2        ; Calculate the Q channel energy
                ADDI        @ACC_Q0,rx_Q
                ADDC        @ACC_Q1,mod_Q
                STI         rx_Q,@ACC_Q0
                STI         mod_Q,@ACC_Q1
                ADDI        @ACCE_Q0,tmpr_2         ; Accummulate the I channel energy
                LDIU        @ACCE_Q1,rx_Q
                ADDC        zero_r,rx_Q
                STI         tmpr_2,@ACCE_Q0
                STI         rx_Q,@ACCE_Q1

unitout:
                LDIU        idx_tbl,mod_I
                LDIU        idx_tbl,mod_Q
                LSH         left16,mod_I

                POP         idx_tbl                 ; Recover idx_tbl reg. value
                POP         ST                      ; Recover status reg. value
                OR          0400h,ST                ; Freeze the cache in order for ISR
                RETI
                .end
```

List 9: TMS320C30 DSP enhanced linearizer program.
File Name: 30p.c30

```
on              .set          1
off             .set          0
linear          .set          on                            ; define this program for linearization
calib           .set          off

                .include      "macro.fcn"                   ; Load the macro function from file
                                                            ; "macro.fcn"
                .include      "const.h30"                   ; load the constant assignment heading
                                                            ; "const.h30"
                .include      "int_vec.h30"                 ; Load the interrupt vector assignment table
                                                            ; "int_vec.h30"
                .include      "memory.h30"                  ; Load the memory assignment heading
                                                            ; "memory.h30"

                .include      "initial.c30"                 ; Load the initialization routine
                                                            ; "initial.c30"

                .text

*
* dsptopc == 'high' => DMA is free; dsptopc == 'low' => DMA is busy
* pctodsp == 'high' => new data; pctodsp == 'low' acknowledge C30 signal
*
Object00:
Comm0:
*
* Check whether the PC has transfer a new table into the dual port memory
*
                TSTB          pctodsp,IOF                   ; Check whether PC has signaled to DSP
                LDIU          @OBJ1PTR,objtmpr              ; Switch to object1 task
                BD            objtmpr
                LDIZ          @COMM0,objtmpa                ; Without new table, next go back to
                                                            ; object0::Comm0
                LDINZ         @COMM1,objtmpa                ; With new table, next go to object::Comm1
                STI           objtmpa,@OBJ0PTR              ; Update object0 pointer

Object01:
Comm1:
*
* A new table arrived, start the DMA transfer and acknowledge to PC
*
                STI           bs_r,*+DMA_adr(DMA_cnt)         ; Enable the DMA
                ANDN          dsptopc,IOF                     ; Assert a 'low' into XF0

Object02:
Comm2:
*
* Check whether the PC acknowledge the DSP signal
*
                TSTB          pctodsp,IOF                   ; ? XF1 == 'low'
                LDIU          @OBJ1PTR,objtmpr              ; Switch to object1 task
                BD            objtmpr
                LDINZ         @COMM2,objtmpa                ; Without Ack from PC, next go back to
                                                            ; object0::Comm2
                LDIZ          @COMM3,objtmpa                ; With Ack from PC, next go to object0::Comm3
                STI           objtmpa,@OBJ0PTR              ; Update object0 pointer

Object03:
Comm3:
*
* Check whether the DMA has finished its transfer
*
                CMPI          *+DMA_adr(DMA_cnt),zero_r     ; Transfer finish ?
                LDIU          @OBJ1PTR,objtmpr              ; not finished, Switch to object1 task
                BD            objtmpr
                LDINZ         @COMM3,objtmpa                ; Not finish transfer, next go back to
                                                            ; object0::Comm3
                LDIZ          @COMM4,objtmpa                ; Finish transfer, next go to object0::Comm4
                STI           objtmpa,@OBJ0PTR              ; Update object0 pointer

Object04:
Comm4:
*
* Send a signal to PC to indicate the transfer is completed and dual port memory is free
*
** Note avoid reg. conflict
                STI           dpadr_r,*+DMA_adr(DMA_src)    ; Set the source dual port address
                OR            dsptopc,IOF                   ; Assert a 'high' into XF0
*               LDIU          @OBJ1PTR,objtmpr              ; Switch to object1 task
                LDIU          @ENERGY1,objtmpr              ; Switch to object1::Energy1
                STI           tbl_adr,*+DMA_adr(DMA_dst)    ; Set the destination table address
                BD            objtmpr
                LDIU          @COMM0,objtmpr                ; Next go to object0::Comm0
                STI           objtmpr,@OBJ0PTR              ; Update object0 pointer
                XOR           blksize,tbl_adr               ; Update table address pointer

Object10:
```

```
Energy0:
            CMPI        @SMP_EN,zero_r              ; Check whether system is in stable condition
            LDIU        @OBJ2PTR,objtmpr            ; Switch to object2 task
            BD          objtmpr
            LDIGT       @ENERGY0,objtmpr            ; If not stable, next go to object1::Energy0
            LDILE       @ENERGY1,objtmpr            ; If stable, next go to object1::Energy1
            STI         objtmpr,@OBJ1PTR            ; Update the object1 pointer

Object11:
Energy1:
            STI         zero_r,@ACCE_I0             ; Set the accumulate energy I variable
                                                    ; (ACCE_I) to zero
            STI         zero_r,@ACCE_I1
            STI         zero_r,@ACCE_Q0             ; Set the accumulate energy Q variable
                                                    ; (ACCE_Q) to zero
            STI         zero_r,@ACCE_Q1
            STI         zero_r,@OUT_CNT             ; Set the symbol counter (OUT_CNT) to zero
            LDIU        one,objtmpa                 ; Load one into objtmpa which is used to
                                                    ; enable the energy computational routine in
                                                    ; ISR
            LDIU        @OBJ2PTR,objtmpr            ; Switch to object2 task
            BD          objtmpr
            STI         objtmpa,@ENG_FLG            ; Enable the energy computational routine in
                                                    ; ISR
            LDIU        @ENERGY2,objtmpr            ; Next go to object1::Energy2
            STI         objtmpr,@OBJ1PTR            ; Update the object1 pointer

Object12:
Energy2:
            LDIU        @EGENADR,objtmpa
            LDIU        @OBJ2PTR,objtmpr
            CMPI        *objtmpa,zero_r
            BD          objtmpr
            LDIZ        @ENERGY3,objtmpr            ; If disable, next go to object1::Energy3
            LDINZ       @ENERGY2,objtmpr            ; If no request, next go the same object task
            STI         objtmpr,@OBJ1PTR            ; Update the object1 pointer

Object13:
Energy3:
            LDIU        @EGENADR,objtmpa            ; Load the Power enable flag address
            STI         zero_r,@ENG_FLG             ; Disable the energy comptation task in ISR
            LDIU        @ACCE_Q1,objtmpr            ; Transfer the accumulated energy of channel
            STI         objtmpr,*++objtmpa(5)       ;         I ,Q and number of symbol counted to
            LDIU        @ACCE_Q0,objtmpr            ;         the dual port memory location for 486
                                                    ;         access
            STI         objtmpr,*--objtmpa
            LDFU        @ACCE_I1,objtmpr
            STF         objtmpr,*--objtmpa
            LDFU        @ACCE_I0,objtmpr
            STF         objtmpr,*--objtmpa
            LDIU        @OUT_CNT,objtmpr
            STI         objtmpr,*--objtmpa
            LDIU        @OBJ2PTR,objtmpr            ; Switch to object2 task
            BD          objtmpr
            STI         set_en,*--objtmpa           ; Acknowledge to 486 that a new set of data
                                                    ; is ready
            LDIU        @ENERGY4,objtmpr            ; Next go to object1::Energy4
            STI         objtmpr,@OBJ1PTR            ; Update the object1 pointer


Object14:
Energy4:
            LDIU        @OBJ2PTR, objtmpr
            BD          objtmpr
            LDIU        @ENERGY4,objtmpr            ; Next go to object1::Energy4
            STI         objtmpr,@OBJ1PTR            ; Update the object1 pointer
            NOP

Object20:
Check0:
            CMPI        @SMP_EN,zero_r              ; if sampling is not finished then branch
            LDIU        @OBJ0PTR,objtmpr            ; Switch to object0 task
            BD          objtmpr
            LDIGE       @CHECK0,objtmpa             ; Not avaible, next go back to
                                                    ; object2::Check0
            LDILT       @CHECK1,objtmpa             ; Avaible, next go to object2::Check1
            STI         objtmpa,@OBJ2PTR            ; Update object2 pointer

Object21:
Check1:
            LDIU        *-dly_adr(nosmpwd*(nospdly-1)+QIo),point0        ; Load the first QIo
            LDIU        point0,point1               ; Load the first QIo
            ASH         right16,point0              ; 2'c Qo
            LSH         left16,point1               ; Find Io
            ASH         right16,point1              ; 2'c Io
            LDIU        zero,objtmpr                ; Reset objtmpr to zero
            MPYI        point0,point0               ; Qo^2
            CMPI        @THSHLD2,point0             ; Check whether greater than threshold value
            ADDC        zero,objtmpr                ; If greater, C=1 and objtmpr += 1
            MPYI        point1,point1               ; Io^2
            CMPI        @THSHLD2,point1             ; Check whether greater than threshold value
            ADDC        zero,objtmpr                ; If greater, C=1 and objtmpr += 1 Check
                                                    ; whether
```

```
                                                        ;         Qo or Io either greater than
                                                        ; threshold value
                LDIU         @OBJ0PTR,objtmpr           ; Switch to object0 task
                BD           objtmpr
                LDINZ        @RE_EN,objtmpa             ; If either of them are greater than the
                                                        ; threshold, next go to object2::Re_en
                LDIZ         @CHECK2,objtmpa            ; If not, next go to object2::Check2
                STI          objtmpa,@OBJ2PTR           ; Update object2 pointer

Object22:
Check2:
                ADDI         point0,point1,idx_s        ; Find Ao^2
                STI          idx_s,*dly_adr             ; Save the Ao^2
                LDIU         idx_s,idx_e                ; Copy Ao^2 to both index regs.
                LSH          idxof_s,idx_s              ; Calculate the 1st lookup index for sampling
                                                        ; task
                LSH          idxof_e,idx_e              ; Calculate the 1st lookup index for envelope
                                                        ; task
                LDIU         @OBJ0PTR,objtmpr           ; Switch to object0 task
                BD           objtmpr
                LDIU         @SMPL0,objtmpa             ; Next go to object2::Smpl0
                STI          objtmpa,@OBJ2PTR           ; Update the object2 pointer
                STI          sets2on,@OBJ2_ON           ; Set the object3_done flag as not finish

Object23:
Smpl0:
*
* Check whether the current sampling data amplitude is out of range or not.
* If 'YES' ignore the entire sampling data and go to object2::wait to check
*               the object3 task finished
* If 'No' next go to object2::calc_s0
*
                CMPI         mapsize,idx_s              ; Check whether out of range
                LDILT        @SMPL1,objtmpa             ; if !(out of range) next go to
                                                        ; object2::smpl1
                BD           Env0                       ; Switch to object3::env0
                LDIGE        @WAIT,objtmpa              ;         else next go to object2::wait
                ADDI         d_offset,idx_s             ; Add offset into idx_s in order for locating
                                                        ; the mapping table address
                STI          objtmpa,@OBJ2PTR

Object24:
Smpl1:
*
* This object2::smpl1 is used to check whether the current sampled data has been placed
* into the sampling table before.
* If 'Yes' the next processing object2::smpl6 will check how many looped samplings has
*               been processed.
* If 'NO' the next processing object2::smpl2 or object2::smpl3 will set the corresponding
*               location flag and transfer the sampling data set to the sampling table.
*
******* Note: tmpr_s1 keeps the corresponding status bit position.
                LDIU         *+map_adr(idx_s),objtmpr   ; Get the 2nd lookup index from the
                                                        ; mapping table
                LDIU         one,tmpr_s1                ; Set tmpr_s1 to one for status memory
                                                        ; checking
                CMPI         sthfsz,objtmpr             ; Substract 2nd index by sthfsz without
                                                        ;         changing the content inside the reg.
                BND          ck_sts0                    ; If C=1, check status memory 0
                LDIU         @OBJ3PTR,objtmpa           ; Switch to object3 task
                AND          stmask,objtmpr             ; Get the bit index for the destinated status
                                                        ; memory
                LSH          objtmpr,tmpr_s1            ; Get the corresponding status bit to
                                                        ; postion.
                TSTB         @SMP_STS1,tmpr_s1          ; Test the corresponding status bit wrt
                                                        ; status memory1
                BD           objtmpa
                LDIZ         @SMPL2,objtmpr             ; if OFF (i.e. not occurpied) next go to
                                                        ; object3::smpl2
                LDINZ        @SMPL6,objtmpr             ;         else next go to object2::smpl6
                STI          objtmpr,@OBJ2PTR           ; Update the object2 pointer
ck_sts0:
                TSTB         @SMP_STS0,tmpr_s1          ; Test the corresponding status bit wrt
                                                        ; status memory0
                BD           objtmpa
                LDIZ         @SMPL3,objtmpr             ; if OFF (i.e. not occurpied) next go to
                                                        ; object2::smpl3
                LDINZ        @SMPL6,objtmpr             ;         else next go to object2::smpl6
                STI          objtmpr,@OBJ2PTR           ; Update the object2 pointer
*
* This object2::smpl2 and object2::smpl3 is used to set the corresponding location flag (64 bit status memory)
* in order to indicate the sampling table location has now been occuriped.
*
Object25:
Smpl2:
                BD           Calc_s0                    ; Go to object2::Calc_s0
                OR           @SMP_STS1,tmpr_s1          ; Set occupation in the status memory1
                STI          tmpr_s1,@SMP_STS1          ; Update the status memory1
                LDIU         *-dly_adr(dlyest*nosmpwd+Qa),objtmpr    ; Get the last sampling data

Object26:
Smpl3:
                BD           Calc_s0                    ; Go to object2::Calc_s0
                OR          @SMP_STS0,tmpr_s1                ; Set occupation in the status memory0
                STI         tmpr_s1,@SMP_STS0                ; Update the status memory0
                LDIU        *-dly_adr(dlyest*nosmpwd+Qa),objtmpr    ; Get the last sampling data

Object27:
Calc_s0:
*
* The following two objects (object2::calc_s0 and object2::calc_s1) are used to adjust the
* delay of the sampling data.
*
                LDIU        *-dly_adr((dlyest+1)*nosmpwd+Qa),tmpr_s1         ; Get the 2nd last
                                                             ; sampling data
                ASH         right16,objtmpr                  ; Get 2'c Qa
                ASH         right16,tmpr_s1                  ; Get 2'c Qa
                SUBI        tmpr_s1,objtmpr                  ; Find the difference
                FLOAT       tmpr_s1                          ; Convert to floating point format
                FLOAT       objtmpr                          ; Convert to floating point format
                MPYF        @D_Qa,objtmpr                    ; Multiply by the slope factor
                ADDF        objtmpr,tmpr_s1                  ; Add to the 2nd last sampling data
                MPYF        @G_Qa,tmpr_s1                    ; Multiply by the gain factor
                FIX         tmpr_s1                          ; Convert back to fix point format
                LSH         left16,tmpr_s1                   ; left shift the adjusted Qa by 16 bits
                LDIU        @OBJ3PTR,objtmpr                 ; Switch to object3 task
                BD          objtmpr
                LDIU        @CALC_S1,objtmpa                 ; Next go to object2::Calc_s1
                STI         objtmpa,@OBJ2PTR                 ; Update object2 pointer
                STI         tmpr_s1,@TMP_S1                  ; Backup tmpr_s1 into TMP_S1

Object28:
Calc_s1:
                LDIU        *-dly_adr(dlyest*nosmpwd+Ia),objtmpr    ; Get the last sampling data
                LDIU        *-dly_adr((dlyest+1)*nosmpwd+Ia),tmpr_s1         ; Get the 2nd last
                                                             ; sampling data
                ASH         right16,objtmpr                  ; Get 2'c Ia
                ASH         right16,tmpr_s1                  ; Get 2'c Ia
                SUBI        tmpr_s1,objtmpr                  ; Find the difference
                FLOAT       tmpr_s1                          ; Convert to floating point format
                FLOAT       objtmpr                          ; Convert to floating point format
                MPYF        @D_Ia,objtmpr                    ; Multiply by the slope factor
                ADDF        objtmpr,tmpr_s1                  ; Add to the 2nd last sampling data
                MPYF        @G_Ia,tmpr_s1                    ; Multiply by the gain factor
                FIX         tmpr_s1                          ; Convert back to fix point format
                AND         lower,tmpr_s1                    ; Get the adjusted Ia
                LDIU        @OBJ3PTR,objtmpr                 ; Switch to object3 task
                BD          objtmpr
** Not use objtmpa because of reg. conflict
                LDIU        @SMPL4,objtmpr                   ; Next go to object2::smpl4
                STI         objtmpr,@OBJ2PTR                 ; Update object2 pointer
                OR          @TMP_S1,tmpr_s1                  ; Mix with Qa into a 32 bits data
Object29:
Smpl4:
*
* This object transfers the current sampling data set to the corresponding sampling table
* position.
** Involve reg. conflict
                LDIU        *+map_adr(idx_s),objtmpa         ; Load the 2nd lookup index
                MPYI        nostwd,objtmpa                   ; Calculate the sample table address
                ADDI        @SMPTADR,objtmpa
                STI         zero_r,@LOOP_S                   ; Reset the looped sampling counter to zero
                STI         tmpr_s1,*++objtmpa
** Note: minus 2 is used since there is one time slot delay between the recevied
* modulating signal and the transmitted signal.
                LDIU        *-map_adr(nosmpwd*(nospdly-2)+Qp),tmpr_s1        ; Get Qp
                LDIU        *-map_adr(nosmpwd*(nospdly-2)+Ip),objtmpr        ; Get Ip
                ANDN        upper,tmpr_s1                    ; Compress Qp,Ip into QIp
                LSH         right16,objtmpr
                OR          objtmpr,tmpr_s1
                LDIU        *map_adr,objtmpr                 ; Get Ao^2 and transfer QIa
                STI         objtmpr,*-objtmpa(1)             ; Transfer Ao^2 and QIp
||              STI         tmpr_s1,*+objtmpa(1)
                LDIU        @OBJ3PTR,objtmpr                 ; Switch to object3 task
                BD          objtmpr
                LDIU        @SMPL5,objtmpa                   ; Next go to object2::smpl5
                STI         objtmpa,@OBJ2PTR                 ; Update the object2 pointer
                LDIU        @SMP_CNT,tmpr_s1                 ; Load the sampling counter value

Object2a:
Smpl5:
*
* Check whether there are enough sampling data in the sampling table
*
                ADDI        one,tmpr_s1                      ; Add the sampling counter by one
                STI         tmpr_s1,@SMP_CNT                 ; Update the sampling counter value
                CMPI        stsize,tmpr_s1                   ; Check whether the sampling table completed
                LDIU        @OBJ3PTR,objtmpr                 ; Switch to object3 task
                BD          objtmpr
                LDIGE       @SMPL7,objtmpa                   ; If (completed) next go to object2::smpl7
                LDILT       @WAIT,objtmpa                    ; If !(completed) next go to object2::wait
                STI         objtmpa,@OBJ2PTR                 ; Update the object2 pointer

Object2b:
Smpl6:
*
```

```
* Check the sampling counter over the threshold.
* If !(over) go to object2::wait to check the object3 task finished
* else increment the loopped sampling counter by one and check this counter whether it is
* over the limit.
* If (over the limit)=>(i.e. over the value of nolpsmp) next go to object2::smpl7 to
*               enable the sampling table transfer.
* If !(over the limit) next go to object2::wait to check the object3 task finished
*
                LDIU        @SMP_CNT,tmpr_sl                ; Load the sampling counter value
                CMPI        nomxsmp,tmpr_sl                 ; Check the sampling counter over the
                                                            ; threshold
                BLTD        Wait                            ; If !(over) go to object2::wait else
                                                            ; continue
                LDIU        @LOOP_S,tmpr_sl                 ; Load the loopped sampling counter value
                ADDI        one,tmpr_sl                     ; Add one into the loopped sampling counter
                LDIU        @OBJ3PTR,objtmpr                ; Switch to object3 task
                STI         tmpr_sl,@LOOP_S                 ; Update the loopped sampleing counter
                CMPI        nolpsmp,tmpr_sl                 ; Check whether the counter is over the limit
                BD          objtmpr
                LDIGE       @SMPL7,objtmpa                  ; If (over) next go to object2::smpl7
                LDILT       @WAIT,objtmpa                   ; If !(over) next go to object2::wait
                STI         objtmpa,@OBJ2PTR                ; Update the object2 pointer

Object2c:
Smpl7:
*
* Enable the sampling table to be accessed by the PC and reset 64 bits status memory,
* sampling counter and loopped sampling counter. Change the sampling table pipeline.
*
******* Involve register conflict in the following instructions
                LDIU        @SMPTADR,objtmpa                ; Load the current sampling table address
                LDIU        @OBJ3PTR,objtmpr                ; Switch to object3 task
                LDIU        @WAIT,tmpr_sl                   ; Next go to object2::wait
                STI         set_en,*+objtmpa(sten_of)       ; Enable the current sampling table
                                                            ; for PC access
                LDIU        *+objtmpa(nxt_st),objtmpa       ; Load the next sampling table
                                                            ; address
                STI         zero_r,@SMP_CNT                 ; Clear sampling counter
                STI         zero_r,@LOOP_S                  ; Clear looped sampling counter
                CMPI        *+objtmpa(sten_of),zero_r       ; Check whether the next sampling
                                                            ; table is occupied
                                                            ;         by the PC
                LDIP        *+objtmpa(nxt_st),objtmpa       ; If 'Yes' load the next 2 sampling
                                                            ; table address
                STI         zero_r,@SMP_STS0                ; Clear 64 bits status memory
                STI         zero_r,@SMP_STS1
                AHEAD       nostbl                          ; Call the macro::AHEAD to insert the command
                                                            ;         [LDIP     *+objtmpa(nxt_st),objtmpa]
                BD          objtmpr
                STI         zero_r,*+objtmpa(sten_of)       ; Lock the sampling table for saving
                                                            ; the new samples
                STI         tmpr_sl,@OBJ2PTR                ; Update the object2 pointer
                STI         objtmpa,@SMPTADR                ; Update the sampling table address pointer

Object2d:
Wait:

*
* Check the ENVELOPE task has been completed.
*               If !(complete) next loop back the same STATE
*               If (complete) next go to re-enable the sampling process
*
                LDIU        @OBJ0PTR,objtmpr                ; Assume go to object0 task
                CMPI        @OBJ2_ON,zero_r                 ; Check whether object3 task finished
                LDINZ       @OBJ3PTR,objtmpr                ; If !(complete) go to object3 task
                BD          objtmpr
                LDINZ       @WAIT,objtmpa                   ; If !(complete) loop back the same object
                LDIZ        @RE_EN,objtmpa                  ; If (complete) next go to object2::re_en
                STI         objtmpa,@OBJ2PTR                ; Update object2 pointer

Object2e:
Re_en:
*
* Re-enable the sampling process
*
                LDIU        @DLY_ST,dly_adr                 ; Reset dly_adr point to delay table
*               LDI         @CH_SEL,objtmpr                 ; Toggle to another channel
*               XOR         @CH4_TOG,objtmpr                ; Not use objtmpa because of reg. conflict
*               STI         objtmpr,@CH_SEL
*               STI         objtmpr,*+ch4_adr(ch4_ctl)
*
* The 1st channel is assumed to channel 0 in 4 channel card
*
                LDIU        @OBJ0PTR,objtmpr                ; Switch to object0 task
                BD          objtmpr
                LDIU        @CHECK0,objtmpa                 ; Next go to object2::check0
                STI         objtmpa,@OBJ2PTR                ; Update the object2 pointer
                STI         zero_r,@SMP_EN                  ; Enable sampling routine again
*
* Envelope task :: Check idx_e at the beginning and clear OBJ2_ON when finished
*

Object30:
```

```
Env0:
*
* Check whether the current sampling data amplitude is out of range or not.
* If 'YES' ignore the entire sampling data and go to object3::env3 to set
*               the object3 task finished
* If 'No' next go to object3::calc_e0
*
                LDIU        @OBJ0PTR,objtmpr                ; Switch to object0 task
                LDILT       @CALC_E0,objtmpa                ; if !(out of range) next go to
                                                            ; object3::calc_e0
                CMPI        mapsize,idx_e                   ; Check whether out of range
                BD          objtmpr
                LDIGE       @ENV3,objtmpa                   ;         else next go to object3::env3
                ADDI        d_offset,idx_e                  ; Add offset into idx_e in order for locating
                                                            ;         the mapping table address
                STI         objtmpa,@OBJ3PTR                ; Update the object3 pointer

Object31:
Calc_e0:
*
* The following three objects (object3::calc_e0 and object3::calc_e2) are used to adjust
* the delay of the sampling data.
*
                LDIU        *-dly_adr(Envlp),objtmpr        ; Get the last sampling data
                LDIU        *-dly_adr(nosmpwd+Envlp),tmpr_s2        ; Get the 2nd last sampling data
                ASH         right20,objtmpr                 ; Get 2'c envelope sampling data
                ASH         right20,tmpr_s2                 ; Get 2'c envelope sampling data
                LDIU        @CH_SEL,objtmpa                 ; Load the channel selected reg.
                CMPI        @CH4_INT,objtmpa                ; Check which channel is going to process
                BNZD        Calc_e1                         ; if demodr, go to object3::calc_e1
                SUBI        tmpr_s2,objtmpr                 ; Get the difference between points
                FLOAT       tmpr_s2                         ; Convert to the float point format
                FLOAT       objtmpr                         ; Convert to the float point format
                BD          Calc_e2                         ; Go to object3::calc_e2
                MPYF        @D_Em,objtmpr                   ; Multiply by the delay factor for modr
                ADDF        objtmpr,tmpr_s2                 ; Adjust the 2nd last sampling data
                MPYF        @G_Em,tmpr_s2                   ; Multiply by the gain factor for modr

Object32:
Calc_e1:
                MPYF        @D_Ed,objtmpr                   ; Multiply by the delay factor for demodr
                ADDF        objtmpr,tmpr_s2                 ; Adjust the 2nd last sampling data
                MPYF        @G_Ed,tmpr_s2                   ; Multiply by the gain factor for demodr

Object33:
Calc_e2:
                FIX         tmpr_s2                         ; Convert to fix point format
                LDIU        @OBJ0PTR,objtmpr                ; Switch to object0 task
                BD          objtmpr
                STI         tmpr_s2,*-dly_adr(Envlp)        ; Update the adjusted sampling data
                LDIU        @ENV1,objtmpr                   ; Next go to object3::env1
                STI         objtmpr,@OBJ3PTR                ; Update the object3 pointer

Object34:
Env1:

Object35:
Env2:

Object36:
Env3:
                LDIU        @OBJ0PTR,objtmpr                ; Switch to object0 task
                BD          objtmpr
                LDIU        @ENV3,objtmpa                   ; Next go to ENV3 ( for safety only )
                STI         zero_r,@OBJ2_ON                 ; Set the object3 operating flag as finish
                STI         objtmpa,@OBJ3PTR                ; Update object3 pointer
*
* Interrupt Service Routine for getting data from
* pi/4 DQPSK modulator and transmitting the predistoted
* I & Q signal to the DAC
*
                .align                                      ; To align the following code into cache

TxRxdata:
                PUSH        ST                              ; Save status reg. into the stack
                PUSH        tmpa_1                          ; Save tmpa_1 reg. into the stack

Sampling:
                LDI         *+adc_adr(adc_chA),rx_I         ; In Chan A,B
||              LDI         *+adc_adr(adc_chB),rx_Q         ; Note: There is a memory conflict.
                STI         mod_I,*+adc_adr(dac_chA)        ; Out Chan A,B
||              STI         mod_Q,*+adc_adr(dac_chB)        ; Note: There is a memory conflict.

*
* The above two instructions must be placed as early as the beginning the ISR, the MAX
* seperation of these instructions from the top of ISR must be less than 20 instruction
* cycles. Note: there are above 8 cycles for acknowledging the interrupt and two wait
* state for accessing the ADC or DAC devices. As the result, there are only about 3 to 4
* single cycle instructions can be performed.
*
                LDI         *c26_adr,tmpa_1                 ; load the new modulated I and Q signals

*
```

```
* This section is used to push the sampled data into the delay buffer when it is enable.
*               The storing sequence is QIo,Qa,Ia,Qp,Ia.
*

SaveDlyBuf:
                STI         mod_I,*+dly_adr(4)
                LDI         one,mod_I                   ; Set tmpr_2 to one
                CMPI        @SMP_EN,zero_r              ; Check the sampling enable condition
                BNZD        Engcal                      ; if not ready to sample then jump to predistion routine
                LDILE       zero,mod_I                  ; if SMP_EN >= 0 then without increase SMP_EN
                ADDI        @SMP_EN,mod_I
                STI         mod_I,@SMP_EN               ; Update SMP_EN value
                CMPI        @DLY_STP,dly_adr            ; Check whether the delay buffer is completely fill
                BLTD        Engcal                      ; if not reach the bottom then jump to 'Next'
                STI         tmpa_1,*dly_adr++(2)
                STI         rx_Q,*-dly_adr(1)           ; Save the current sampling data into the
||              STI         mod_Q,*+dly_adr(1)          ;         delay buffer
                STI         rx_I,*dly_adr++(3)
                STI         dpadr_r,@SMP_EN             ; if reach the bottom then disable sampling
*
* The section is used to compute the energy of the sampled I and Q channel signals.
* (Note: three commands inside this section are used for pre-calculate the LUT index
*               rather than compute the energy.)
*

Engcal:
                LDI         @ENG_FLG,mod_I              ; Check whether the power computation routine
                                                        ; is enable?
                                                        ; If ENG_FLG=1, calculate the power
                BZD         prdstn                      ; If ENG_FLG=0, not calculate the power

                ADDI        @OUT_CNT,mod_I
                ASH         right16,rx_I
                ASH         right16,rx_Q

                STI         mod_I,@OUT_CNT              ; Save the updated symbol count variable.

                MPYI        rx_I,rx_I                   ; Calculate the I channel energy
                ADDI        @ACCE_I0,rx_I               ; Accummulate the I channel energy
                LDIU        @ACCE_I1,mod_I
                ADDC        zero_r,mod_I

                MPYI        rx_Q,rx_Q                   ; Calculate the Q channel energy
                ADDI        @ACCE_Q0,rx_Q               ; Accummulate the I channel energy
                LDIU        @ACCE_Q1,mod_Q
                ADDC        zero_r,mod_Q

                STI         rx_I,@ACCE_I0               ; Backup the accummulated energy for power
                                                        ; computation
                STI         mod_I,@ACCE_I1              ;         in the external task.
                STI         rx_Q,@ACCE_Q0
                STI         mod_Q,@ACCE_Q1

*
* This section is used to compute the LUT index and predistort the modulated signals
*
prdstn:
                LDIU        tmpa_1,mod_Q
                LSH         left16,tmpa_1               ; Convert the input modulated 16bits Qo to
                                                        ; 32bits
                ASH         right16,tmpa_1
                ASH         right16,mod_Q               ; Convert the input modulated 16bits Qo to
                                                        ; 32bits

                FLOAT       tmpa_1,rx_I                 ; Convert the 2's complement values Io to
                                                        ; floating point values.
                FLOAT       mod_Q,rx_Q                  ; Convert the 2's complement values Qo to
                                                        ; floating point values.

                MPYI        tmpa_1,tmpa_1               ; Calculate Io^2

                MPYI        mod_Q,mod_Q                 ; Calculate Qo^2
                ADDI        mod_Q,tmpa_1                ; Calculate Ao^2

                LSH         tuncat,tmpa_1               ; Calculate the table index
                LSH         left1,tmpa_1
                ADDI        tbl_adr,tmpa_1              ; Calculate the look up address for beta

                MPYF        *tmpa_1,rx_I,mod_I          ; Compute Ip
                MPYF        *+tmpa_1(1),rx_Q,mod_Q
                SUBF        mod_Q,mod_I
                MPYF        *tmpa_1,rx_Q,mod_Q          ; Compute Qp
                MPYF        *+tmpa_1(1),rx_I
                ADDF        rx_I,mod_Q

                FIX         mod_I                       ; Convert to fix point format
                FIX         mod_Q                       ; Convert to fix point format

                POP         tmpa_1                      ; Recover tmpa_1 reg. value
                POP         ST                          ; Recover status reg. value
                OR          0400h,ST                    ; Freeze the cache in order for ISR
                RETI
                .end
```

List 10: 486 PC C++ header files for the enhanced linearizer.
File Name: Format.h, Const.h, Cinterfa.h, Clinearl.h, Clineari.h, Ctest0.h, Ctest1.h, Utility.h, Map_cntr.h.

```
// File Name: format.h

#ifndef type_def
#define type_def
typedef long double ldouble;
typedef   unsigned long       ulong;
typedef   unsigned short      ushort;
typedef unsigned char         uchar;
#endif

#ifndef ulong_data
#define ulong_data
union ULDATA {
    ulong    ulword;
    struct {
          unsigned char byte0;
          unsigned char byte1;
          unsigned char byte2;
          unsigned char byte3;
          } of;
    struct {
          ushort usword0;
          ushort usword1;
          } of_;
    struct {
          short sword0;
          short sword1;
          } of__;
    };
#endif

//////////////////////////////////////////////////////////////////////////////////////////

// File Name : Const.h

#ifndef   const_H
#define   const_H

#define   C30PROG   "30p.out"
#define C30CAL                "30c.out"

#define NOLOOP                500
#define RUNTYPE               'n'
#define ONE                   65536

#define MAX_TIME_CAL          60        // The max allowed time (in second) for calibration.
//  If this requirement can not be satisfied within this time (i.e. the calibation
//  process can not converge to an allowed tolerance level), the entire linearization
//  object will treat as invalid.

#define TIME_CAL              5         // Time (in second) for the calibration.
//  Note: in this 5 seconds calibration, there are more 25000 samples for evaluating
//  the time delay constant.

// Maximum numbers of board is 16
#define NO_BOARDS   4
#define BOARD_0     0x290
#define BOARD_1     0x390
#define BOARD_2     0x690
#define BOARD_3     0x790
#define BOARD_ADDRESS         {BOARD_0, BOARD_1, BOARD_2, BOARD_3}

#define   NO_OF_SAMPLES       3
#define   SMP_SIZE  64
#define   SMP_SIZE_1          63
#define   SMP_SIZE1 65
#define   SMP_SIZE2 66
#define   SMP_SIZE3 67
#define   TL_SMP_SIZE         192                 // NO_OF_SAMPLES*SMP_SIZE
#define   TL_SMP_SIZE1        193                 // NO_OF_SAMPLES*SMP_SIZE+1
#define   TL_SMP_SIZE2        194                 // NO_OF_SAMPLES*SMP_SIZE+2
#define   BLK_SIZE  0x800
#define   HLF_SIZE  0x400

#define   BLK0                0x8d0
#define   BLK1                0xd0
#define   ALPHA_BLK0          0x8d0
#define   BETA_BLK0 0xcd0

#define G_Ia                  0x660
#define G_Qa                  0x661
#define G_Em                  0x662
#define G_Ed                  0x663
#define D_Ia                  0x664
#define D_Qa                  0x665
```

```
#define D_Em                 0x666
#define D_Ed                 0x667

#define   ALPHA_DUAL_ADR     0x30000
#define   BETA_DUAL_ADR      0x30400
#define   ETEN               0x30800

#define ENERGY_ENABLE        0x30801
#define NO_SYMBOL  0x30802
#define ENERGY_I0  0x30803
#define ENERGY_I1  0x30804
#define ENERGY_Q0  0x30805
#define ENERGY_Q1  0x30806
#define ACCS_I0              0x30807
#define ACCS_I1              0x30808
#define ACCS_Q0              0x30809
#define ACCS_Q1              0x3080A
#define DEST_EN              0x3080B
#define DEST_I               0x3080C
#define DEST_Q               0x3080D
#define   SMPTBL_ST          0x3086C

#define   MAX_MAG2           536384378           // From simulation and experiment
#define   LUT_DMAG2          524288              // Because of the tuncation in lookup table,
                                                 //     this value should be 2^19=524288
#define LUT_HLF_DMAG2        262144              // Half of the LUT_DMAG2

#define MTBL_DMAG2           2097152             // Because of the tuncation in sampling table,
                                                 //     this value should be 2^21=2097152


#define TWO_PWR32            4294967296.0
#define MAX_OUTPUT_MAG2      4.16204663163E18    // The maximum DAC output magnitude square
                                                 //  = (90%*2^15*ONE)^2
                                                 // There is 10% tolerance of max output.
#define MAX_DAC_OUTPUT       32767

// From the simulation.
#define UNIT_REF_POWER       259342663.508       //  The averge total output power level of unit output
#define UNIT_REF_CH_PWR      129671331.754       //  The averge total output power level of unit output
                                                 //         on each channel. (i.e. channel I and channel Q)

#define UNIT_REF_POWER_RATIO 1.5                 // The ratio of the actual output power wrt the unit
                                                 //         output power level.
#define CLIPPING_ADJ_RATIO   0.8                 // Reduce the entire LUT magnitude by this ratio when the
                                                 //  clipping occured.

#define DELAY_TOLERANCE      0.002     // 0.2% tolerance for estimating time delay constant
                                       // If out of this range, a continuous calibration will
                                       // be performed.

#define PWR_REG_TOLERANCE    0.002     //  .2% tolerance wrt the ref_power_level
#define CLP_REG_TOLERANCE    0.002     //  .2% tolerance wrt the ref_output_level

#define DETECT_THRESHOLD     25        // The numbers of wait cycle for the
                                       // response from the C30 board. This is
                                       // used in the Cinterface class to detect
                                       // whether the specified C30 board is avaiable
                                       // or not.

#define REG_SW_THSHLD        10        // The threshold value to switch regulators
// NOTE : The regulators are the power regulator and the clipping regulator.

#define OFFSET_THSHLD        546       // The threshold voltage when the calibrated offset
                                       //    voltage is considered in the computation.
                                       // value in voltage/3*32768 (i.e. 0.05)
#define   W_new              0.0625    // 1/16
#define   W_old              0.9375    // 15/16

#define PWR_DAMPING          0.0625    // must be less than or equal to W_new
#define OUTPUT_DAMPING       0.0625    //  otherwise the clipping will happen.

#define OCCUPIED             0xffffffffL
#define NOT_USED             0x0L
#define STOP_ENG_TASK        0x0L
#define SET_ENABLE           0x30000L
#define NOSTBL               10

#define NO_INIT              0
#define WORKING              1
#define STOP                 2

#define TICKPERSEC           18.2

#endif

////////////////////////////////////////////////////////////////////////////////////////

// File Name : Cinterfa.h

#include "format.h"
#include "const.h"

#ifndef Cinterface_h
```

```
#define Cinterface_h

class Cinterface {

#define BIT0 0x0001
#define BIT1 0x0002
#define BIT2 0x0004
#define BIT3 0x0008
#define BIT4 0x0010
#define NOT0 0xFFFE
#define NOT1 0xFFFD
#define NOT2 0XFFFB
#define NOT3 0XFFF7
#define NOT4 0XFFEF

protected:
    enum MEMTYPE {DUAL,ALL};
    enum {OFF,ON};

private:
    union UTAG{
        float ieeerep;
          long transrep;
          };

    union USDATA{
          ushort usword;
          struct    {
              unsigned char byte0;
              unsigned char byte1;
              } of;
          };

    struct SECTIONS {
          ulong adre;
          ulong len;
          ulong disp;
          };  // an array of structures used if COFF files are loaded

protected:

    uchar  board_status;  //  Board setup status Reg.
    ushort board_ID;            //      Board accessing address.

private:
    ushort ControlReg;          //      Control register image used by functions
    ushort C30port;   //      I/O Base Address.
    ushort commreg;   //      Comm. Reg. low half Address
    ushort hicommreg;           //      Comm. Reg. high half Address
    ushort memadrreg;           //      Memory address reg address
    ushort himemadrreg;   //  high byte of memory address reg adr
    ushort controlregadr; //  Ctrl. Reg. Address
    ushort statusregadr;  //  Status Reg. Address

private:

    ushort count;
    UTAG   pat;
    USDATA nosect, ustemp;
    ULDATA ultemp;

// Note: 'ultemp' and 'ustemp' are shared by other member functions.
//         BE CAREFUL the order !!

private:

    static ushort boards_condition;
    uchar  set_board(const ushort &boardadr);

protected:

// Function which sets the base adr variable used by all other functions
// Returns I/O base address if selection is successful, returns current base
// address if called with param =1. If called with an invalid address it rets
// 0 and prints an error msg on the PC Must always use this function in your
// source code.
    Cinterface::Cinterface(const ushort &boardadr);

// Freeze the entire C30 board without doing anything.
    void Freeze_C30();

// Pulse Reset bit high to low.
    void Reset(void);

// Take Reset bit high.
    void AssertReset(void);

// Take Reset bit low.
    void RelReset(void);

// Halt DSP program execution.
    void Hold(void);
// Restart DSP program execution from halt.
    void UnHold(void);

// Enable interface address counter.
    void CntrEnb(void);

// Disable interface address counter.
    void CntrDis(void);

// Set address change direction to inc.
    void AutoInc(void);

// See if Held: low = held, high = running
    uchar Held(void);

// asserts hold and waits for ack
    void HoldAndWait(void);

// releases hold and waits for ack
    void UnHoldAndWait(void);

// puts address into port address registers
    void SetAddr(const ulong &adr);

// Convert an IEEE ldouble value to 320C30 format and put to board memory
    void PutFloat(const ulong &adr, const MEMTYPE &memty, const ldouble &fmemdata);

// Convert an IEEE ldouble value to 320C30 format and put to dual port memory
    void PutFloat(const ulong &adr, const ldouble &fmemdata);

//convert an array of ldouble values to 320C30 representation then put into board dual port mem
    ulong WrBlkFlt(const ulong &adr, const ushort &nofval, const ldouble *arry);

//convert an array of ldouble values to 320C30 representation then put into board mem
    ulong WrBlkFlt(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, const ldouble *arry);

//convert a constant of ldouble values to 320C30 representation then put into board dual port mem
    ulong WrBlkFlt(const ulong &adr, const ushort &nofval, const ldouble &constant);

//convert a constant of ldouble values to 320C30 representation then put into board mem
    ulong WrBlkFlt(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, const ldouble &constant);

// Put a single 32 bit code into a mem loc
    void Put32Bit(const ulong &adr, const MEMTYPE &memty, const ulong &inst);

// Put a single 32 bit code into a dual port mem loc
    void Put32Bit(const ulong &adr, const ulong &inst);

// Put a block of 32 bit code into a dual port mem loc from an array
    ulong WrBlk32(const ulong &adr, const ushort &nofval, const ulong *arry);

// Put a block of 32 bit code into a board mem loc from an array
    ulong WrBlk32(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, const ulong *arry);

// Put a block of 32 bit code into a dual port mem loc from a constant
    ulong WrBlk32(const ulong &adr, const ushort &nofval, const ulong &constant);

// Put a block of 32 bit code into a board mem loc from a constant
    ulong WrBlk32(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, const ulong &constant);

// Pull a 320C30 value from mem, perform conversion and return as a long double value
    ldouble GetFloat(const ulong &adr,const MEMTYPE &memty);

// Pull a 320C30 value from dual port mem, perform conversion and return as a long double value
    ldouble GetFloat(const ulong &adr);

// Pull a 320C30 value from mem, perform no conversion and return as a 32 bit pattern
    ulong Get32Bit(const ulong &adr, const MEMTYPE &memty);

// Pull a 320C30 value from dual port mem, perform no conversion and return as a 32 bit pattern
    ulong Get32Bit(const ulong &adr);

// Read a block of 32 bit codes from dual port mem into an array
    ulong RdBlk32(const ulong &adr, const ushort &nofval, ulong *arry);

// Read a block of 32 bit codes from board mem into an array
    ulong RdBlk32(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, ulong *arry);

// Read a block of 32 bit codes from dual port mem into a ULDATA array
    ulong RdBlk32(const ulong &adr, const ushort &nofval, ULDATA *arry);

// Read a block of 32 bit codes from board mem into a ULDATA array
    ulong RdBlk32(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, ULDATA *arry);

// Download a COFF format file into board memory
// Returns zero if sucessful, 1 or 2 if an error occurs.
    uchar LoadObjectFile(const char *file_name);

    void pctodsp(const uchar &signal);

    uchar dsptopc();
};
#endif
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////

// File Name : Clinearl.h

#include "format.h"
#include "const.h"
#include "Cinterfa.h"

#ifndef Clinearlib_h
#define Clinearlib_h

class Clinearlib:protected Cinterface {

protected:

    ULDATA  *sample_array;
    ldouble *Ao2,*alpha,*beta,*table;
    ldouble gain_factor;
    ulong   last_smptbl;

    ldouble output_power,acc_power;
    ulong   no_output_counted;
    ldouble energy_I, energy_Q;

    uchar   regulator_switch;

// NOTE : 1. The regulators are the power regulator and the clipping regulator.
//        2. This regulator_switch variable is used to keep track when should
//                 the other regulator be tured on.

private:

    ldouble Ip,Qp,Ia,Qa,Aa2;
    ldouble dAo2,mag2,dmag2,delta_F;
    ldouble max_mag2,max_output_mag2,output_mag2;

    ldouble gain_adj_I,gain_adj_Q,offset_I, offset_Q;
    ldouble dly_coff_I,dly_coff_Q;
    ldouble ref_power_level,inv_ref_power_level;
    ldouble ref_output_level,inv_ref_output_level;
    ldouble power_error_ratio, output_error_ratio;

    ULDATA  *ptr_sample;
    ldouble *alpha_slope,*beta_slope;
    ldouble *ptr_alpha, *ptr_beta, *ptr_alpha_slope, *ptr_beta_slope;
    ldouble *ptr_Ao2,*ptr_table;

    ulong   smptbl_st,smptbl_end,sten_adr;
    ushort  count;

public:
    Clinearlib(const ushort &board_no);
    ~Clinearlib();

// Start the C30 program
    void    start();

// Restart the C30 program
    void    set_for_linearization();

// Calibrate the feedback circuit
    uchar   calibrating();

// Check whether a new set of samples data is ready. if ready then set
//       the signal flag and get a new set of samples.
    uchar   get_samples();

// Check whether the accumulate energy values for channel I and Q are ready.
//       if ready then signal to C30 to stop the energy compution task and
//       load the most update energy values.
    uchar   get_energy();

// To regulate the output power to a specific level if no clipping occcurs
    void    power_regulator();

// Convert and adjust the new sampling data to error vector .
    void    data_coversion();

// Calculate the predistorted coefficients
    void    find_slope();

// Compute the look-up table
    void    compute_table();

// Update the look-up table to C30 board
    void    update_table();

// Update the look-up table to C30 board with UNITY
    void    update_table(const float &p_alpha, const float &p_beta);

    friend  class Ctest0;
```

```
};
#endif
////////////////////////////////////////////////////////////////////////////////////////////////////

// File Name : Clineari.h

#include "format.h"
#include "Clinearl.h"
#include "const.h"

#ifndef Clinearizer_h
#define Clinearizer_h

class Clinearizer:protected Clinearlib

{

public:

    enum STATE_STATUS {
           START_LINEARIZATION,
           GET_SAMPLES,
           GET_ENERGY,
           DATA_CONVERSION,
           FIND_SLOPE,
           COMPUTE_TABLE,
           UPDATE_UT,
           UPDATE_LUT,
           CHECK_ENDING,
           QUIT);

    Clinearizer(const ushort &board_no, const ulong &noloop);

    virtual uchar linearization();

    // Return the board setup status
    uchar setup_status();
    ulong loop_counted();

    friend class Ctest1;

protected:

    ulong counter;
    STATE_STATUS state_reg;
    ulong loopcounter;

private:

    uchar calibration();

};

class CDUlinearizer:public Clinearizer

{

public:

    CDUlinearizer(const ushort board_no=BOARD_0, const ulong noloop=NOLOOP);

    uchar linearization();

    friend class Ctest1;

};

class CDlinearizer:public Clinearizer

{

public:

    CDlinearizer(const ushort board_no=BOARD_0, const ulong noloop=NOLOOP);

    uchar linearization();

    friend class Ctest1;

};

class CNDUlinearizer:public Clinearizer

{

public:

    CNDUlinearizer(const ushort board_no=BOARD_0, const ulong noloop=NOLOOP);

    uchar linearization();

    friend class Ctest1;
```

```
};

#endif

//////////////////////////////////////////////////////////////////////////////////////

// File Name : Ctiming.h

#include "format.h"
#include "const.h"

#ifndef Ctiming_H
#define Ctiming_H

class Ctiming {

private:
    long bstart, bend, c_time,i_time;

public:
    enum TIME { SETTIME, SHOWTIME, SETSTART, CHECKTIME};
    uchar timer(const unsigned control, const float duration=0,
                   const ulong numoftrial=0);
};

#endif

//////////////////////////////////////////////////////////////////////////////////////

// File Name : Ctest0.h

#include "format.h"
#include "const.h"
#include "Clinearl.h"

#ifndef Ctest0_h
#define Ctest0_h

class Ctest0

{

private:
    Clinearlib &const linearlib;

public:
    Ctest0(Clinearlib &testing);

    void condition();

// Download samples into default file SAMPLES.TXT
    uchar last_samples(const char *filename="sample.txt");

// Download lookup table into default file INTER.TXT
    uchar last_table(const char *filename="inter.txt");

// Download the adjusting vector into default file vector.TXT
    uchar adjusting_vector(const char *filename="vector.txt");

// Download slope table into default file slope.TXT
    uchar show_slope(const char *filename="slope.txt");
};
#endif

//////////////////////////////////////////////////////////////////////////////////////

// File Name : Ctest1.h

#include "format.h"
#include "const.h"
#include "Clineari.h"

#ifndef Ctest1_h
#define Ctest1_h

class Ctest1
{
private:
    Clinearizer &const linearizer;

public:
    Ctest1(Clinearizer &testing);

    void condition();

// Display the no_symbol_counted, ene:gy_I, energy_Q and output_power onto
//          the screen.
    void show_power();

// Down samples into default file SAMPLES.TXT
    uchar last_samples(con:t char *filename="sample.txt");
```

```
// Down table into default file INTER.TXT
    uchar last_table(const char *filename="inter.txt");
};
#endif

//////////////////////////////////////////////////////////////////////////////////////////

// File Name : utility.h

#include "format.h"
#include "const.h"
#include "Clineari.h"
#include "Ctiming.h"
#include "Ctest1.h"

#ifndef utility_H
#define utility_H

struct PROGARG
{
    ulong noloop;
    char  run_type;
};

int  invalid(unsigned int size);
void retrieve_arguments(const int no_of_arg, char **const arg_value,
                              PROGARG *const progarg);
uchar assign_objects(Clinearizer **const Linearizer, Ctest1 **const Check,
            Ctiming **const timing, uchar *const Linearizer_status,
            const PROGARG *const prog_arg);

#define NoOfUnknown 6
void matrix_mod(double *I, double *Q, double *m, double *unknown, double *matrix);
void matrix_demod(double *I, double *Q, double *m, double *unknown, double *matrix);
uchar gauss_jordan(double *coff, double *result);

#endif

//////////////////////////////////////////////////////////////////////////////////////////

// File Name : map_cntr.h

#ifndef map_cntr_h
#define map_cntr_h
#define MAP_CENTER {\
          8.5,\
          19.5,\
          25,\
          29.5,\
          34,\
          38.5,\
          42.5,\
          46.5,\
          50.5,\
          54.5,\
          59,\
          63.5,\
          68,\
          73,\
          78.5,\
          84,\
          89,\
          93.5,\
          97,\
          100.5,\
          104,\
          106.5,\
          108.5,\
          111,\
          113.5,\
          115.5,\
          117.5,\
          119,\
          120.5,\
          122.5,\
          124.5,\
          126.5,\
          128.5,\
          130.5,\
          132.5,\
          134.5,\
          136.5,\
          138.5,\
          141,\
          144,\
          146.5,\
          149,\
          152,\
          155.5,\
          158.5,\
          161,\
          164,\
          167,\
          170,\
          172.5,\
          175,\
          178,\
          181,\
          183.5,\
          186,\
          189,\
          192,\
          195,\
          198.5,\
          202.5,\
          207,\
          212.5,\
          219.5,\
          240\
}
#endif
```

List 11: 486 PC C++ program files for the enhanced linearizer.
File Name: Cinterfa.cpp.

```
#include <stdio.h>
#include <conio.h>
#include <float.h>
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <string.h>
#include "Cinterfa.h"

//////////////////////////////////////////////////////////////////////////////
//
//  Five marco functions are defined bleow.
//
//////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////////
//
//  DUAL_PORT_LOCATION(ADDRESS)
//
//  Purpose         :  Restrict the input address to within dual port.
//  Parameters      :  ADDRESS of the converted address.
//  Return          :  Dual port address.

#define DUAL_PORT_LOCATION(ADDRESS) ((ADDRESS & 0x3ffff)|0x30000)

//////////////////////////////////////////////////////////////////////////////
//
//  CONVERT_IEEEtoTMS(IEEEfloat)
//
//  Purpose         :  Convert IEEE format float value to TMS320C30 format.
//  Parameters      :  IEEE floating value IEEEfloat.
//  Return          :  ultemp.ulword = the converted TMS320C30 floating format.
//

#define CONVERT_IEEEtoTMS(IEEEfloat)                                         \
    pat.ieeerep=IEEEfloat;                                                   \
    if(pat.transrep==0)                                                                \
          ultemp.ulword=0x80000000;                                          \
    else                                                                     \
          if(pat.transrep&0x80000000) {                                      \
              ultemp.ulword=-(pat.transrep|0x800000);                        \
              if(ultemp.ulword&0x800000)                                               \
                    ultemp.ulword=(ultemp.ulword&0xFFFFFF)|                  \
                         (((pat.transrep^0x40000000)&0x7F800000)<<1);     \
              else                                                           \
                    ultemp.ulword=((ultemp.ulword|0x800000)&0xFFFFFF)|    \
                         (((pat.transrep-0x3F800000)&0x7F800000)<<1);     \
              }                                                                        \
          else                                                                         \
              ultemp.ulword=(pat.transrep&0x7fffff)|                         \
                    (((pat.transrep-0x3F800000)&0x7F800000)<<1);

//////////////////////////////////////////////////////////////////////////////
//
//  CONVERT_TMStoIEEE(TMSfloat)
//
//  Purpose         :  Convert IEEE format float value to TMS320C30 format.
//  Parameters      :  IEEE floating value IEEEfloat.
//  Return          :  pat.ieeerep = the converted TMS320C30 floating format.
//

#define CONVERT_TMStoIEEE(TMSfloat)                                               \
    pat.transrep=TMSfloat.ulword;                                                 \
    if(TMSfloat.of.byte3==0x80)                                                             \
          pat.transrep=0;                                                                   \
    else                                                                          \
          if(TMSfloat.of.byte2&0x80) {                                            \
              TMSfloat.ulword=-((TMSfloat.ulword&0x7fffff)|0x1000000);            \
              if(TMSfloat.of.byte2&0x80)                                                    \
                    pat.transrep=(TMSfloat.ulword&0x807fffff)|                    \
                         (((pat.transrep+0x7f000000)&0xff000000)>>1);  \
              else                                                                \
                    pat.transrep=(TMSfloat.ulword&0x807fffff)|                    \
                         (((pat.transrep+0x80000000)&0xff000000)>>1);  \
              }                                                                             \
          else                                                                              \
              pat.transrep=(TMSfloat.ulword&0x7fffff)|                            \
                    (((pat.transrep+0x7f000000)&0xff000000)>>1);

//////////////////////////////////////////////////////////////////////////////
//
//  GET_USDATA(VARIABLE)
//
//  Purpose         :  Get an unsigned short value a input file stream
//  Parameters      :  VARIABLE an unsigned short variable.
//  Return          :  VARIABLE an unsigned short variable.
```

```
//

#define GET_USDATA(VARIABLE)              \
    VARIABLE.of.byte0=input_file.get(); \
    VARIABLE.of.byte1=input_file.get();

//////////////////////////////////////////////////////////////////////////////
//
//  GET_ULDATA(VARIABLE)
//
//  Purpose         :  Get an unsigned long value a input file stream
//  Parameters      :  VARIABLE an unsigned long variable.
//  Return          :  VARIABLE an unsigned long variable.
//

#define GET_ULDATA(VARIABLE)              \
    VARIABLE.of.byte0=input_file.get(); \
    VARIABLE.of.byte1=input_file.get(); \
    VARIABLE.of.byte2=input_file.get(); \
    VARIABLE.of.byte3=input_file.get();


//////////////////////////////////////////////////////////////////////////////
//
//  Define class static variable.
//
//////////////////////////////////////////////////////////////////////////////


// Initialize the static member variable boards_condition to zero

ushort Cinterface::boards_condition=0;

//////////////////////////////////////////////////////////////////////////////
//
//  Class member functions are defined bleow.
//
//////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////////
//
//  set_board(const ushort &boardadr)
//
//  Purpose         :  Check the corresponding C30 board condition and
//                     set the C30 board enable if this board is available.
//  Parameters      :  boardadr = C30 board address.
//  Return          :  0 not available, 1 available for the C30 board at the
//                     address boardadr.
//
uchar Cinterface::set_board(const ushort &boardadr)
{

    ushort board_choices[]=BOARD_ADDRESS;

// Check the specific C30 board address whether it is matched the system
//        address, and check whether this address has already occupied by
//        other boards. If any one of two conditions does not satisfied,
//        a zero (i.e. not avaiable for the system) is return.

    for (uchar count=0; count<NO_BOARDS; count++)
          if (boardadr==board_choices[count]) break;
    if ((count==NO_BOARDS)||(boards_condition & ((ushort)1 << count)))
          return 0;

// In herer, you can add some of the detecting routine to check whether
//        the C30 board is physically availiable for the system. Now, the
//        this program uses the responds from the hold and unhold to detect
//        whether the card is available or not.

    count=0;
    ControlReg&=NOT1;                          // ensure bit 1 low
    outpw(controlregadr,ControlReg);  // put word out to I/O port.
    while((inpw(statusregadr)&BIT1))
    {
          count++;
          if (count>DETECT_THRESHOLD)
              return 0;
    }
    count=0;
    ControlReg|=BIT1;                          // ensure bit 1 hi
    outpw(controlregadr,ControlReg);  // Put word out to I/O port.
    while(!(inpw(statusregadr)&BIT1))
    {
          count++;
          if (count>DETECT_THRESHOLD)
              return 0;
    }
    boards_condition |= (ushort)1 << count;
    return 1;
}   // End of member function.

//////////////////////////////////////////////////////////////////////////////
```

```
//
//  Cinterface(const ushort &boardadr)
//
//  Class Cinterface constructor. Set new I/O base adr. and update actual
//  port address variables.
//
//  Parameters       : Board address (Integer multiple of 16).
//  Returns          : board_status is set to the corresponding condition of
//                     the C30 board.
//                     0 = C30 board insuccessfully sets up.
//                     1 = C30 board successfully sets up.
//  Action           : Updates all member variables that hold current
//                     I/O addresses for the board registers.
//
Cinterface::Cinterface(const ushort &boardadr)
{

//  Initialize the address variables.

    ControlReg         = 0;
    C30port            = boardadr;
    controlregadr = C30port + 8;
    statusregadr  = C30port + 8;

//  Check validity of the desired base address

    if (set_board(boardadr))
    {

//  Initialize the address variables.

          commreg       = C30port + 0;
          hicommreg   = C30port + 2;
          memadrreg   = C30port + 4;
          himemadrreg = C30port + 6;

          Freeze_C30();
          board_status=1;
          board_ID=boardadr;
    }
    else
          board_status=0;
}   // End of Cinterface's constructor.


//////////////////////////////////////////////////////////////////////////////
//
//  Freeze_C30()
//
//  This member function is used to halt the entire C30 borad.
//
//  Parameters       : Void.
//  Return           : Void.
//
void Cinterface::Freeze_C30()
{
    UnHold();
    RelReset();
    for(count=0;count<8000;count++);
    AssertReset();
    RelReset();
    Put32Bit(01,ALL,0x000C01);
    Put32Bit(0x000C01,ALL,0x600000C0);
    inpw(C30port+0xC);                         // clear interrupts into known state
    Reset();
}   // End of member function.

//////////////////////////////////////////////////////////////////////////////
//
//   Memory Read/Write member functions
//
//////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////
//
//  PutFloat()
//
//  Convert IEEE format ldouble value to TMS320C30 format and put into memory.
//
//  Parameters       :  address to write into, memtype and value to write.
//  Returns           :  0 if success, -1 if fails.
//
void Cinterface::PutFloat(const ulong &adr, const MEMTYPE &memty, const ldouble &fmemdata)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrDis();
          CONVERT_IEEEtoTMS((float) fmemdata)
          outpw(commreg,ultemp.of_.usword0);
          outpw(hicommreg,ultemp.of_.usword1);
          UnHoldAndWait();
    }
    else
          PutFloat(adr, fmemdata);
}   //    End of member function

//////////////////////////////////////////////////////////////////////////////
//
//  PutFloat()
//  Convert IEEE format ldouble value to TMS320C30 format and put into memory.
//
//  Parameters        :  address to write into and value to write.
//  Returns           :  void.
//
void Cinterface::PutFloat(const ulong &adr, const ldouble &fmemdata)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrDis();
    CONVERT_IEEEtoTMS((float)fmemdata)
    outpw(commreg,ultemp.of_.usword0);
    outpw(hicommreg,ultemp.of_.usword1);
}   //    End of member function.

//////////////////////////////////////////////////////////////////////////////
//
//  WrBlkFlt()
//
//  Get IEEE format ldouble values from an array, convert to TMS320C30 format
//  then put onto dual port memory.
//
//  Parameters        :  address to put to,  no of floats to put, array name.
//  Returns           :  The last address + 1.
//
ulong Cinterface::WrBlkFlt(const ulong &adr, const ushort &nofval, const ldouble *arry)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrEnb();
    AutoInc();
    for(count=0; count<nofval; count++)
    {
          CONVERT_IEEEtoTMS((float) *(arry++))
          outpw(commreg,ultemp.of_.usword0);
          outpw(hicommreg,ultemp.of_.usword1);
    }
    return (nofval+adr);
}   //    End of member function

//////////////////////////////////////////////////////////////////////////////
//
//  WrBlkFlt()
//  Get IEEE format ldouble values from an array, convert to TMS320C30 format
//  then put onto board memory.
//
//  Parameters        :  address to put to,  no of floats to put, array name.
//  Returns           :  The last address + 1.
//
ulong Cinterface::WrBlkFlt(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, const ldouble *arry)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrEnb();
          AutoInc();

          for(count=0; count<nofval; count++)
          {
              CONVERT_IEEEtoTMS((float) *(arry++))
              outpw(commreg,ultemp.of_.usword0);
              outpw(hicommreg,ultemp.of_.usword1);
          }
          UnHoldAndWait();
    }
    else
          WrBlkFlt(adr, nofval, arry);
    return (nofval+adr);
}   //    End of member function

//////////////////////////////////////////////////////////////////////////////
//
//  WrBlkFlt()
//
//  Get IEEE format ldouble values from a constant, convert to TMS320C30
//  format then put onto dual port memory.
//
//  Parameters        :  address to put to,  no of floats to put, array name.
//  Returns           :  The last address + 1.
//
ulong Cinterface::WrBlkFlt(const ulong &adr, const ushort &nofval, const ldouble &constant)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrEnb();
    AutoInc();
    CONVERT_IEEEtoTMS((float) constant)
    for(count=0; count<nofval; count++)
    {
          outpw(commreg,ultemp.of_.usword0);
          outpw(hicommreg,ultemp.of_.usword1);
    }
    return (nofval+adr);
}   //    End of member function

////////////////////////////////////////////////////////////////////////////////
//
//  WrBlkFlt()
//
//  Get IEEE format ldouble values from a constant, convert to TMS320C30
//  format then put onto board memory.
//
//  Parameters        :  address to put to,  no of floats to put, array name.
//  Returns           :  The last address + 1.
//
ulong Cinterface::WrBlkFlt(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, const ldouble &constant)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrEnb();
          AutoInc();

          CONVERT_IEEEtoTMS((float) constant)
          for(count=0; count<nofval; count++)
          {
              outpw(commreg,ultemp.of_.usword0);
              outpw(hicommreg,ultemp.of_.usword1);
          }
          UnHoldAndWait();
    }
    else
          WrBlkFlt(adr, nofval, constant);
    return (nofval+adr);
}   //    End of member function

////////////////////////////////////////////////////////////////////////////////
//
//  Put32Bit()
//
//  Deposits a 32 bit value into a memory location
//
//  Parameters        :  address to write into, memtype, and value to write.
//  Returns           :  flag, 0= success, -1 = failure.
//
void Cinterface::Put32Bit(const ulong &adr,const MEMTYPE &memty,const ulong &inst)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrDis();
          ultemp.ulword = inst;
          outpw(commreg,ultemp.of_.usword0);
          outpw(hicommreg,ultemp.of_.usword1);
          UnHoldAndWait();
          }
    else
          Put32Bit(adr, inst);
}   //    End of member function

////////////////////////////////////////////////////////////////////////////////
//
//  Put32Bit()
//
//  Deposits a 32 bit value into a memory location
//
//  Parameters       :  address to write into and value to write.
//  Returns          :  void.
//
void Cinterface::Put32Bit(const ulong &adr, const ulong &inst)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrDis();
    ultemp.ulword = inst;
    outpw(commreg,ultemp.of_.usword0);
    outpw(hicommreg,ultemp.of_.usword1);
}   //    End of member function

////////////////////////////////////////////////////////////////////////////////
//
//  Function  WrBlk32()
//
//  Get 32 bit pattens from an array and put into dual port mem.
//
//  Parameters       :  address to put to, type of mem,no of values to put, array name.
//  Returns          :  The last address + 1.
//
```

```
ulong Cinterface::WrBlk32(const ulong &adr, const ushort &nofval, const ulong *arry)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrEnb();
    AutoInc();
    for(count=0; count<nofval; count++)
    {
          ultemp.ulword = *(arry++);
          outpw(commreg,ultemp.of_.usword0);
          outpw(hicommreg,ultemp.of_.usword1);
    }
    return (nofval+adr);
}    //   End of member function

//////////////////////////////////////////////////////////////////////////////
//
//  WrBlk32()
//
//  Get 32 bit pattens from an array and put into board mem.
//
//  Parameters       :  address to put to, type of mem,no of values to put, array name.
//  Returns          :  The last address + 1.
//
ulong Cinterface::WrBlk32(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, const ulong *arry)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrEnb();
          AutoInc();
          for(count=0; count<nofval; count++)
          {
               ultemp.ulword = *(arry++);
               outpw(commreg,ultemp.of_.usword0);
               outpw(hicommreg,ultemp.of_.usword1);
          }
          UnHoldAndWait();
    }
    else
          WrBlk32(adr, nofval, arry);
    return (nofval+adr);
}    //   End of member function

//////////////////////////////////////////////////////////////////////////////
//
//  WrBlk32()
//
//  Get 32 bit pattens from constant and put into dual port mem.
//
//  Parameters       :  address to put to, type of mem,no of values to put, const.
//  Returns          :  The last address + 1.
//
ulong Cinterface::WrBlk32(const ulong &adr, const ushort &nofval, const ulong &constant)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrEnb();
    AutoInc();
    ultemp.ulword = constant;
    for(count=0; count<nofval; count++)
    {
          outpw(commreg,ultemp.of_.usword0);
          outpw(hicommreg,ultemp.of_.usword1);
    }
    return (nofval+adr);
}   // End of memeber function

//////////////////////////////////////////////////////////////////////////////
//
//  WrBlk32()
//
//  Get 32 bit pattens from constant and put into board mem.
//
//  Parameters       :  address to put to, type of mem,no of values to put, const.
//  Returns          :  The last address + 1.
//
ulong Cinterface::WrBlk32(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, const ulong &constant)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrEnb();
          AutoInc();
          ultemp.ulword = constant;
          for(count=0; count<nofval; count++)
          {
               outpw(commreg,ultemp.of_.usword0);
               outpw(hicommreg,ultemp.of_.usword1);
          }
          UnHoldAndWait();
    }
    else
```

```
          WrBlk32(adr, nofval, constant);
    return (nofval+adr);
}    //   End of member function

/////////////////////////////////////////////////////////////////////////////
//
//  GetFloat()
//
//  Get TMS320C30 format float value from memory and convert to IEEE ldouble
//  format.
//  Parameters          :  address to get from, memtype.
//  Returns             :  The value taken from DSP memory (in IEEE).
//
ldouble Cinterface::GetFloat(const ulong &adr, const MEMTYPE &memty)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrDis();
          ultemp.of_.usword0=inpw(commreg);
          ultemp.of_.usword1=inpw(hicommreg);
          CONVERT_TMStoIEEE(ultemp)
          UnHoldAndWait();
          return ((ldouble)pat.ieeerep);
    }
    else
          return (GetFloat(adr));
}   // End of member function

/////////////////////////////////////////////////////////////////////////////
//
//  GetFloat()
//
//  Get TMS320C30 format float value from dual port memory and convert to
//  IEEE ldouble format.
//
//  Parameters        :  address to get from.
//  Returns            :  The value taken from DSP memory (in IEEE).
//
ldouble   Cinterface::GetFloat(const ulong &adr)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrDis();
    ultemp.of_.usword0=inpw(commreg);
    ultemp.of_.usword1=inpw(hicommreg);
    CONVERT_TMStoIEEE(ultemp)
    return ((ldouble)pat.ieeerep);
}   // End of member function

/////////////////////////////////////////////////////////////////////////////
//
//  Function  Get32Bit()
//
//  Get TMS320C30 value from memory as a bit pattern performs NO conversion
//  just passes this back as a long int.
//
//  Parameters        :  address to get from,       memtype.
//  Returns            :  The value taken from DSP memory.
//
ulong Cinterface::Get32Bit(const ulong &adr,const MEMTYPE &memty)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrDis();
          ultemp.of_.usword0=inpw(commreg);
          ultemp.of_.usword1=inpw(hicommreg);
          UnHoldAndWait();
          return ultemp.ulword;
    }
    else
          return (Get32Bit(adr));
}   //    End of member function


/////////////////////////////////////////////////////////////////////////////
//
//  Get32Bit()
//
//  Get TMS320C30 value from dual port memory as a bit pattern performs NO
//  conversion just passes this back as a long int.
//
//  Parameters        :  address to get from.
//  Returns           :  void.
//
ulong Cinterface::Get32Bit(const ulong &adr)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrDis();
    ultemp.of_.usword0=inpw(commreg);
    ultemp.of_.usword1=inpw(hicommreg);
    return ultemp.ulword;
}   //    End of member function

////////////////////////////////////////////////////////////////////////////////
//
//  RdBlk32()
//
//  Reads a block of 32bit values from dual port memory into an array.
//
//  Parameters      :  address to get from, no of opcodes to get, array name.
//  Returns         :  The last address + 1.
//
ulong Cinterface::RdBlk32(const ulong &adr, const ushort &nofval, ulong *arry)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrEnb();
    AutoInc();
    for(count=0; count<nofval; count++)
    {
          ultemp.of_.usword0=inpw(commreg);
          ultemp.of_.usword1=inpw(hicommreg);
          *(arry++)=ultemp.ulword;
    }
    return (nofval+adr);
}    // End of member function

////////////////////////////////////////////////////////////////////////////////
//
//  RdBlk32()
//
//  Reads a block of 32bit values from dual port memory into an array.
//
//  Parameters      :  address to get from, no of opcodes to get, array name.
//  Returns         :  The last address + 1.
//
ulong Cinterface::RdBlk32(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, ulong *arry)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrEnb();
          AutoInc();
          for(count=0; count<nofval; count++)
          {
              ultemp.of_.usword0=inpw(commreg);
              ultemp.of_.usword1=inpw(hicommreg);
              *(arry++)=ultemp.ulword;
          }
          UnHoldAndWait();
    }
    else
          return RdBlk32(adr, nofval, arry);
    return (nofval+adr);
}    // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  RdBlk32()
//
//  Reads a block of 32bit values from dual port memory into an array.
//
//  Parameters      :  address to get from, no of opcodes to get, array name.
//  Returns         :  The last address + 1.
//
ulong Cinterface::RdBlk32(const ulong &adr, const ushort &nofval, ULDATA *arry)
{
    SetAddr(DUAL_PORT_LOCATION(adr));
    CntrEnb();
    AutoInc();
    for(count=0; count<nofval; count++)
    {
          arry->of_.usword0=inpw(commreg);
          (arry++)->of_.usword1=inpw(hicommreg);
    }
    return (nofval+adr);
}    // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  Function  RdBlk32()
//
//  Reads a block of 32bit values from dual port memory into an array.
//
//  Parameters      :  address to get from, no of opcodes to get, array name.
//  Returns         :  The last address + 1.
//
ulong Cinterface::RdBlk32(const ulong &adr, const MEMTYPE &memty, const ushort &nofval, ULDATA *arry)
{
    if(memty==ALL)
    {
          HoldAndWait();
          SetAddr(adr);
          CntrEnb();
          AutoInc();
          for(count=0; count<nofval; count++)
          {
              arry->of_.usword0=inpw(commreg);
              (arry++)->of_.usword1=inpw(hicommreg);
          }
          UnHoldAndWait();
    }
    else
          RdBlk32(adr, nofval, arry);
    return (nofval+adr);
}    // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  LoadObjectFile()
//
//  Loads a COFF file (xxx.OUT) from the PC memory onto the TMS320C30 Board.
//
//  Parameters : file_name (pointer to string)
//
//  Returns    : 0 = File loaded O.K.
//                 1 = Unable to open file
//                 2 = Invalid address
//
uchar Cinterface::LoadObjectFile(const char *file_name)
{
    ulong loopcount, bytecntr, load_address;

    ifstream input_file(file_name,ios::in|ios::nocreate|ios::binary);

    if (!input_file) return 1;
    input_file.ignore(2);
    GET_USDATA(nosect)
    loopcount = nosect.usword;
    SECTIONS *sects = new SECTIONS [nosect.usword];
    input_file.ignore(12);
    GET_USDATA(ustemp)

    if (ustemp.usword)
    {
          input_file.ignore(30);
          bytecntr=48;
    }
    else
    {
          input_file.ignore(2);
          bytecntr=20;
    }
    for (ushort cursect=0; loopcount!=0; loopcount--,cursect++,bytecntr+=40)
    {
          input_file.ignore(12);
          GET_ULDATA(ultemp)
          sects[cursect].adre = ultemp.ulword;
          GET_ULDATA(ultemp)
          sects[cursect].len  = ultemp.ulword;
          GET_ULDATA(ultemp)
          sects[cursect].disp = ultemp.ulword;
          input_file.ignore(16);
    }

    for (cursect=0; nosect.usword!=0; cursect++, nosect.usword--)
    {
          if(sects[cursect].disp==bytecntr)
          {
              load_address=sects[cursect].adre;
              HoldAndWait();
              SetAddr(load_address);
              CntrEnb();
              AutoInc();
              for(loopcount=sects[cursect].len; loopcount!=0; loopcount--)
              {
                    GET_ULDATA(ultemp)
                    bytecntr+=4;
                    outpw(commreg,ultemp.of_.usword0);
                    outpw(hicommreg,ultemp.of_.usword1);
                    if((++load_address)>0xFFFFFF)
                    {
                        delete [] sects;
                        return 2;
                    }
              }
              UnHoldAndWait();
          }
    }
    delete [] sects;
    return 0;
}  // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  Held()
```

```
//
//  Return indication of if processor held.
//
//  Parameters       : none.
//  Returns          : 0 = Hold line is hi (inactive, i.e. processor running).
//                     1 = Hold line is lo (active, i.e. processor is held).
//
inline uchar Cinterface::Held()
{
   return ((inpw(statusregadr)&2) ? 0 : 1);
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  Reset()
//
//  Cause board to reset.
//
//  Parameters       : none.
//  Returns          : Void
//
void Cinterface::Reset()
{
   AssertReset();
   RelReset();
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  AssertReset()
//
//  Cause reset pin to hi and hold it there.
//
//  Parameters       : none.
//  Returns          : Void
//
inline void Cinterface::AssertReset()
{
   ControlReg|=BIT0;                          // ensure bit 0 hi (assert reset)
   outpw(controlregadr,ControlReg);  // put word out to I/O port.
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  RelReset()
//
//  Release reset.
//  Parameters       : none.
//  Returns          : Void
//
inline void Cinterface::RelReset()
{
   ControlReg&=NOT0;                          // ensure bit 0 low (reset release)
   outpw(controlregadr,ControlReg);  // put word out to I/O port.
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  Hold()
//
//  Cause execution of code on board to be halted.
//
//  Parameters       : none.
//  Returns          : Void
//
inline void Cinterface::Hold()
{
   ControlReg&=NOT1;                          // ensure bit 1 low
   outpw(controlregadr,ControlReg);  // put word out to I/O port.
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  UnHold()
//
//  Cause recomencement of program after a Hold.
//  Parameters       : none.
//  Returns          : Void
//
inline void Cinterface::UnHold()
{
   ControlReg|=BIT1;                          // ensure bit 1 hi
   outpw(controlregadr,ControlReg);  // put word out to I/O port.
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  CntrEnb()
//
//  Enable interface port address counter.
//
//  Parameters       : none.
//  Returns          : Void
```

```
//
inline void Cinterface::CntrEnb()
{
   ControlReg&=NOT2;                          // ensure bit 2 low
   outpw(controlregadr,ControlReg);  // put word out to I/O port.
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  CntrDis()
//
//  Disable interface port address counter.
//
//  Parameters      : none.
//  Returns         : Void
//
inline void Cinterface::CntrDis()
{
    ControlReg|=BIT2;                          // ensure bit 2 hi
    outpw(controlregadr,ControlReg);  // put word out to I/O port.
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  AutoInc()
//
//  Put interface port address counter to increment mode.
//
//  Parameters      : none.
//  Returns         : Void
//
inline void Cinterface::AutoInc()
{
   ControlReg&=NOT3;                          // ensure bit 3 low
   outpw(controlregadr,ControlReg);  // put word out to I/O port.
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  HoldAndWait()
//
//  Cause execution of code on board to be halted, waits for processor to
//  ack the hold request.
//
//  Parameters      : none.
//  Returns         : Void
//
void Cinterface::HoldAndWait()
{
   ControlReg&=NOT1;                          // ensure bit 1 low
   outpw(controlregadr,ControlReg);  // put word out to I/O port.
   while((inpw(statusregadr)&BIT1));
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  UnHoldAndWait()
//
//  Cause recomencement of program after a Hold, waits for processor to
//  indicate unheld.
//
//  Parameters      : none.
//  Returns         : Void
//
void Cinterface::UnHoldAndWait()
{
   ControlReg|=BIT1;                          // ensure bit 1 hi
   outpw(controlregadr,ControlReg);  // Put word out to I/O port.
   while(!(inpw(statusregadr)&BIT1));
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  SetAddr()
//
//  Puts an address into the interface area address ports.
//
//  Parameters      : the 24 bit address.
//  Returns         : Void
//
inline void Cinterface::SetAddr(const ulong &adr)
{
    ultemp.ulword=adr;
    outpw(memadrreg,ultemp.of_.usword0);
    outpw(himemadrreg,ultemp.of_.usword1);
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  pctodsp()
//
//  Set the XOF1 flag to either 1 or 0.
//
```

```
//  Parameters      : the 8 bit data.
//  Returns         : Void
//
void Cinterface::pctodsp(const uchar &signal)
{
    if (signal)
          ControlReg |= BIT4;
    else
          ControlReg &= NOT4;
    outpw(controlregadr,ControlReg);
}   // End of member function.

/////////////////////////////////////////////////////////////////////////////
//
//  dsptopc()
//
//  Check the XOF0 flag whether 1 or 0.
//
//  Parameters      : void.
//  Returns         : ON or OFF
//
uchar Cinterface::dsptopc()
{
    return ((inpw(statusregadr)&BIT3) ? ON : OFF);
}   // End of member function.
```

List 12: 486 PC C++ program files for the enhanced linearizer.
File Name: Clinearl.cpp.

```
#include <stdlib.h>
#include <math.h>
#include <iostream.h>
#include "Clinearl.h"
#include "map_cntr.h"
#include "Ctiming.h"

////////////////////////////////////////////////////////////////////////////////
//
//  CONVERT_Int64toLdouble(upper32,lower32,ldouble_var)
//
//  Purpose         : Convert the signed 64 bit int to long double value.
//  Parameters      : The upper and lower 32 bit data and the ldouble variable.
//  return          : The converted ldouble value in the ldouble_var.
//

#define CONVERT_Int64toLdouble(upper32,lower32,ldouble_var)      \
    {                                                                        \
    ulong tmp_upper32=upper32;                                               \
    ldouble_var = (tmp_upper32 & 0x80000000) ?                           \
                      -(TWO_PWR32*(ldouble)(~tmp_upper32)+(ldouble)(~lower32)+1):\
                      TWO_PWR32*(ldouble)tmp_upper32+(ldouble)lower32;  \
    }

////////////////////////////////////////////////////////////////////////////////
//
//  Note: you can use class Ctest0 to test this class member function. The
//        following codes show how to use Ctest0.
//
//  Ctest0 check(*this);              // Set up the link for class Ctest0
//  check.last_table("chk_int0.txt");   // Call the Ctest0 member functions
//  check.last_table();                         // Use the default setting
//  check.last_samples("chk_smp0.txt");
//  check.adjusting_vector("chk_vec0.txt");
//

////////////////////////////////////////////////////////////////////////////////
//
//  Clinearlib(const ushort &board_no)
//
//  This class constructor is used to initialize its base class (Cinterface).
//  If the base class can be initialized to the specific C30 board, this
//  constructor will load the calibration program into the C30, and perform
//  the initialization for this class.
//

Clinearlib::Clinearlib(const ushort &board_no):Cinterface(board_no)
{

//  Check whether C30 calibration program can be successfully download into
//  the board.

    if (Cinterface::board_status)
          Cinterface::board_status=!LoadObjectFile(C30CAL);

//  Check whether this board is avaiable.

    if (Cinterface::board_status)
    {

//  Initialize the acc_power to zero
          acc_power=0;

//  Initialize the required gain factor and gain adjusting factor to "1".
//  Note: gain_factor has been normalized to 2^16=65536 (i.e. ONE=65536).

          gain_factor=ONE;
          gain_adj_I=gain_adj_Q=1;

//  Initialize the offset_I and offset_Q to zero

          offset_I=offset_Q=0;

//  Get the original delay coefficients of both channel that set in the C30
//  program.

          dly_coff_I=GetFloat(D_Ia,Cinterface::ALL);
          dly_coff_Q=GetFloat(D_Qa,Cinterface::ALL);

// Enable the power computation routine once the C30 board output is stable.

          Put32Bit(ENERGY_ENABLE,SET_ENABLE);

// Signal to the C30 DMA controller that no table is needed to transfer.

          pctodsp(Cinterface::OFF);
    }
```

```
}   // End of Clinearlib's constructor.

//////////////////////////////////////////////////////////////////////////////
//
// ~Clinearlib()
//
// This class destructor is used to release the memory location that create
// for the variables sample_array, Ao2, alpha, beta, alpha_slope, beta_slope
// and table.
//

Clinearlib::~Clinearlib()
{
    if (Cinterface::board_status)
    {
          delete [] sample_array;
          delete [] Ao2;
          delete [] alpha;
          delete [] beta;
          delete [] alpha_slope;
          delete [] beta_slope;
          delete [] table;
    }

}   // End of Clinearlib's destructor.

//////////////////////////////////////////////////////////////////////////////
//
//  set_for_linearization()
//
//  This member function is used to initialize the entire the linearization
//  system. Procedures include "download the C30 linearization program into
//  C30", "download the new calibrated parameters into C30 program", and
//  "initialize all the required parameters and variables".
//

void Clinearlib::set_for_linearization()
{

    Freeze_C30();
    LoadObjectFile(C30PROG);

//  Create a list of dynamic array variables.

    sample_array = new ULDATA [TL_SMP_SIZE1];
    Ao2 = new ldouble [SMP_SIZE2];
    alpha = new ldouble [SMP_SIZE2];
    beta = new ldouble [SMP_SIZE2];
    alpha_slope = new ldouble [SMP_SIZE1];
    beta_slope = new ldouble [SMP_SIZE1];
    table  = new ldouble [BLK_SIZE];

//  Calculate the power and output reference level.

    ref_power_level = UNIT_REF_POWER*UNIT_REF_POWER_RATIO;
    ref_output_level = MAX_OUTPUT_MAG2; //MAX_MAG2;
    inv_ref_power_level = 1/ref_power_level;
    inv_ref_output_level = 1/ref_output_level;
    acc_power=0;

//  Initialize the regulator
//  if regulator < REG_SW_THSHLD then using power regulator first
//  if regulator >= REG_SW_THSHLD then using output level regulator.

    regulator_switch=0;
//  regulator_switch=REG_SW_THSHLD;

//  Initialize the sampling table enable address, the first and the one behine
//  the first one sampling table addresses locations.
//  Note: smptbl_st = the location of the first sampling table in the cirrular
//                        single linked sampling table buffer (i.e. table 1)
//           last_smptbl = the location of the last sampling table  (i.e. table NOSTBL)

    sten_adr=(smptbl_st=SMPTBL_ST)+TL_SMP_SIZE1;
    last_smptbl=(NOSTBL-1)*(TL_SMP_SIZE2)+smptbl_st;

//  Initialize the Ao2 array
//  Define the center values of the Ao2. Without this value, the first set
//  of sampling data may crash the linearization because there may have a
//  chance of getting zero values in the sampling values.
//

    ldouble map_center[]=MAP_CENTER, *tmp_map_ctr;

    for(count=0, ptr_alpha=alpha, ptr_beta=beta, ptr_Ao2=Ao2,
          tmp_map_ctr=map_center; count<SMP_SIZE2; count++)
    {
          *(ptr_alpha++) = ONE;
          *(ptr_beta++) = 0;
          *(ptr_Ao2++)= (count==0) ? -5*(ldouble)MAX_MAG2 :
                          (count==SMP_SIZE1) ? 10*(ldouble)MAX_MAG2 :
                          MTBL_DMAG2* (*(tmp_map_ctr++));
    }

//  Initialize the generated lookup table to be the ideal value
//  (i.e. alpha=1, beta=0)

    for(ptr_table=table, count=0; count<HLF_SIZE; count++)
    {
          *(ptr_table++)=ONE;
          *(ptr_table++)=0;
    }

//  Initialize the first block lookup table to be the ideal value.
//  (i.e. alpha=1, beta=0)

    WrBlkFlt(ALPHA_BLK0, Cinterface::ALL, BLK_SIZE, table);

//  Set the calibrated values into the C30 linearization program.

    PutFloat(D_Ia, Cinterface::ALL, dly_coff_I);
    PutFloat(D_Qa, Cinterface::ALL, dly_coff_Q);
    PutFloat(G_Ia, Cinterface::ALL, gain_adj_I);
    PutFloat(G_Qa, Cinterface::ALL, gain_adj_Q);

//  Set the last sampling table to be occupied. This is used to guarante
//  that C30 will not overwrite the first sampling table if the C30
//  works much fast than the 486. (Note: the probablity of this case is
//  very small since the C30 fill 10 sampling tables before 486 starts
//  to check whether the first sampling table is avaible. But it sometimes
//  happens.

    Put32Bit((last_smptbl+TL_SMP_SIZE1), OCCUPIED);

//  Enable the power computation routine once the C30 board output is stable.

    Put32Bit(ENERGY_ENABLE,SET_ENABLE);

//  Signal to the C30 DMA controller that no table is needed to transfer.

    pctodsp(Cinterface::OFF);

}   // End of member function.

///////////////////////////////////////////////////////////////////////////////
//
// start()
//
// This member function is used to reset the C30 DSP and start the C30
// program.
//

void Clinearlib::start()

{
    Reset();
}   // End of member function.

///////////////////////////////////////////////////////////////////////////////
//
// calibrating()
//
// This member function is used to obtain the energy level of channel I
// and Q, and compute the average output power level in order to calibrate
// the output imbalance of I and Q channels
//

uchar Clinearlib::calibrating()
{

    ldouble old_dly_coff_I,old_dly_coff_Q;

//  Create and initialize the timer variables for counting the calibrating
//  time.

    Ctiming *calib_timer=new Ctiming;
    Ctiming *total_calib_timer=new Ctiming;
    total_calib_timer->timer(Ctiming::SETSTART);

    start();

    do
    {
          cout << "Calibrating !!!" << endl;

// Store the previous delay coefficients

          old_dly_coff_I=dly_coff_I;
          old_dly_coff_Q=dly_coff_Q;

//  Timing the calibation and waiting for finish

          calib_timer->timer(Ctiming::SETSTART);
          while(!calib_timer->timer(Ctiming::CHECKTIME,TIME_CAL,0));
//  Signal to the delay computation to stop the current delay computation
//  and wait for C30 to acknowledge.

          Put32Bit(DEST_EN,SET_ENABLE);
          while (Get32Bit(DEST_EN));

//  Get the most update time delay coefficients

          dly_coff_I=GetFloat(DEST_I);
          dly_coff_Q=GetFloat(DEST_Q);

//  Check whether the new set of coefficients within a pre-define tolerance
//  and time out for calibation.

    } while(
      ((fabsl((dly_coff_I-old_dly_coff_I)/old_dly_coff_I)>DELAY_TOLERANCE)||
       (fabsl((dly_coff_Q-old_dly_coff_Q)/old_dly_coff_Q)>DELAY_TOLERANCE))&&
       (Cinterface::board_status=!
          total_calib_timer->timer(Ctiming::CHECKTIME,MAX_TIME_CAL,0)));

//  Check the previous calibation whether successful or not. If successful,
//  compute the gain imbalance and the dc offset.

    if(Cinterface::board_status)
    {

//  Signal to stop the energy and signal accumulation routines, wait for
//  C30 acknowledging and get the update energy values for both channels.

          Put32Bit(ENERGY_ENABLE,STOP_ENG_TASK);
          while(!get_energy());

//  Get both channel's accumulated signals and convert it from 63 signed integer
//  to ldouble number format.

          CONVERT_Int64toLdouble(Get32Bit(ACCS_I1),Get32Bit(ACCS_I0),offset_I)
          CONVERT_Int64toLdouble(Get32Bit(ACCS_Q1),Get32Bit(ACCS_Q0),offset_Q)

//  Compute the new dc offset values

          offset_I=offset_I/no_output_counted;
          offset_Q=offset_Q/no_output_counted;

//  Check whether the dc offset value is above or below the threshold values
//  if the new offset values higer than the threshold, the offset is
//  considered as part of gain imbalance computation, else the offset
//  is set to zero and gain imbalance is independed on the offset.

          if (fabsl(offset_I)>OFFSET_THSHLD)
          {
              gain_adj_I=powl((energy_I/no_output_counted-offset_I*offset_I)/
                          UNIT_REF_CH_PWR,0.25);
              offset_I = -offset_I/(gain_adj_I+1);
              gain_adj_I=1/gain_adj_I;
          }
          else
          {
              offset_I=0;
              gain_adj_I=powl(UNIT_REF_CH_PWR*no_output_counted/
                          energy_I,0.25);
          }

          if (fabsl(offset_Q)>OFFSET_THSHLD)
          {
              gain_adj_Q=powl((energy_Q/no_output_counted-offset_Q*offset_Q)/
                          UNIT_REF_CH_PWR,0.25);
              offset_Q = -offset_Q/(gain_adj_Q+1);
              gain_adj_Q=1/gain_adj_Q;
          }
          else
          {
              offset_Q=0;
              gain_adj_Q=powl(UNIT_REF_CH_PWR*no_output_counted/
                          energy_Q,0.25);
          }
    }

    cout << dly_coff_I << endl
           << dly_coff_Q << endl
           << gain_adj_I << endl
           << gain_adj_Q << endl;

//  Erase all the temperate timing variables.

    delete calib_timer;
    delete total_calib_timer;

//  Return the board status value

    return Cinterface::board_status;

}   // End of member function.
```

```
////////////////////////////////////////////////////////////////////////////////
//
//  get_samples()
//
//  This member function is used to get a new set of sampled data from
//  the single linked cirrular sampling tables (SL sampling tables)
//  whenever a new set of sampled data is ready. Besides, this
//  member function also stops the C30's energy computation routine
//  if a new set of sampled data is successfully downloaded.
//  Note: 486 keeps one of the SL sampling tables in order to indicate
//  to the C30 where the most recent sampling table the 486 has
//  occupied. Once a new set of sampling data is arrived, the 486
//  will occupied the new sampling table and release the last holded
//  sampling table to C30 board.
//
//  Return : 0 if the sampling table is not ready.
//            1 if the sampling table is ready.
//  (Note: the variables of smptbl_st, last_smptbl, sten_adr and sample_array
//          will be updated if return value = 1)
//

uchar Clinearlib::get_samples()
{

//  Test whether the current sampling table is ready or not.
//        (i.e. Check whether a new set of sampled data is ready.)

    if(Get32Bit(sten_adr))
    {
// Occupied the current sampling table if a new set of sampled data is ready
        Put32Bit(sten_adr,OCCUPIED);

//  Erase the Ao^2 values in last sampling table to zero and also release
//        this table to C30 board.

        for (count=0; count<SMP_SIZE; count++,last_smptbl+=3)
            Put32Bit( last_smptbl, NOT_USED);
            Put32Bit( ++last_smptbl, NOT_USED);

//  Load the new sampled data from the current sampling table from the
//        dual port memory.

        RdBlk32(smptbl_st, TL_SMP_SIZE1, sample_array);

//  Save the current sampling table location as for the last sampling
//        table location and get the next sampling table location and the
//        sampling enable flag location.

        last_smptbl=smptbl_st;
        smptbl_st = sample_array[TL_SMP_SIZE].ulword;
        sten_adr = smptbl_st+TL_SMP_SIZE1;

// Disable the energy computation task in the C30 in order to get the most
//        update output energy level of channel I and Q.

        Put32Bit(ENERGY_ENABLE,STOP_ENG_TASK);

// Return 1 if successfully load the new set of sampled data

        return 1;
    }
    else

// Return 0 as the sampling table is not ready.

        return 0;

}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  get_energy()
//
//  This member function is used to get the most update accumulate energy
//  values of channel I and Q and compute the output power of the
//  predistorted signal.
//  Note: (1) 486 first checks whether C30 has been acknowledged the stop
//  request and put the new values into the dual port memory. If
//  it is ready (i.e. ENERGY_ENABLE flag > 0), then load the most
//  update values from the desinated dual port memory locations.
//  (2) 486 needs not re-enable the C30's energy computation task
//  after downloading a new values. Because this energy
//  computation task will be re-enable by the C30's COMM4 task.
//  (i.e. After a new LUT table has been update to the C30 board.)
//
//  return : 0 if the C30 board has not acknowledged
//             1 new accumulated energy values has been download and
//               a new output power level is also computed.
//  (Note: the member variables of energy_I, energy_Q, output_power,
//          no_output_counted are also update if the return values = 1)
//
```

```
uchar Clinearlib::get_energy()
{

// Check whether C30 has been acknowledged.

    if (Get32Bit(ENERGY_ENABLE))
    {

// If C30 has been acknowledged then Get the numbers of symbol counted in
//        the last output interval, and the accumulated energy level of
//        channel I and Q.

// compute the output power using the new getting data.

          energy_I=(ldouble)Get32Bit(ENERGY_I0)+
                     (ldouble)Get32Bit(ENERGY_I1)*TWO_PWR32;
          energy_Q=(ldouble)Get32Bit(ENERGY_Q0)+
                     (ldouble)Get32Bit(ENERGY_Q1)*TWO_PWR32:
          no_output_counted=Get32Bit(NO_SYMBOL);
          acc_power+=(output_power=
          gain_adj_I*gain_adj_I*((energy_I/no_output_counted)-offset_I*offset_I)+
          gain_adj_Q*gain_adj_Q*((energy_Q/no_output_counted)-offset_Q*offset_Q));

// return 1 to indicate a new set of data or values that has been updated
//        and computed.

          return 1;
    }

// Indicate that C30 has not acknowledged to the 486 request.

    return 0;

}  // End of member function.

/////////////////////////////////////////////////////////////////////////////
//
//  power_regulator()
//
//  This member function is used to check whether the output power is
//  limited within the specific tolerance. If 'Yes', a new gain_factor
//  will be calculated.
//

void Clinearlib::power_regulator()
{
    power_error_ratio=(output_power-ref_power_level)*inv_ref_power_level;
    if (fabsl(power_error_ratio)>PWR_REG_TOLERANCE)
    {
          gain_factor*=(1-PWR_DAMPING*power_error_ratio);
          cout << "Power regulator in use" << endl;
    }
}   // End of member function.

/////////////////////////////////////////////////////////////////////////////
//
//  data_coversion()
//
//  This member function is used to convert the sampling data to the error
//  vector (alpha and beta) values and adjust these vector by the gain
//  factor.
//

void Clinearlib::data_coversion()
{

// Set the first alpha value of error vector tc gain factor since this
//        vector is corresponding to Ao2=-5*Aomax which is the starting
//        point of the linear interpolation.

    ptr_alpha=alpha;
    *(ptr_alpha++)=gain_factor;

    for(ptr_sample=sample_array, ptr_beta=beta+1, ptr_Ao2=Ao2+1,
          count=0; count<SMP_SIZE; count++)
    {

//  If the received Ao2!=0.0 then get new Sp and Sa, and compute
//      a new set of alpha and beta coefficients.

          if (ptr_sample->ulword)
          {
               *(ptr_Ao2++)=(ptr_sample++)->ulword;
               Ia = ptr_sample->of__.sword0;
               Qa = (ptr_sample++)->of__.sword1;
               Ip = ptr_sample->of__.sword0;
               Qp = (ptr_sample++)->of__.sword1;
               Aa2 = Ia*Ia+Qa*Qa;
               *(ptr_alpha++) = gain_factor*(Ia*Ip+Qa*Qp)/Aa2;
               *(ptr_beta++) = gain_factor*(Ia*Qp-Ip*Qa)/Aa2;
          }
          else
          {
```

```
//  If Ao2==0.0, then keep Ao2 (i.e. keep the previous Ao2 value),
//        and assign alpha and beta as the previous set of error vector
//        except the first one. For the first error vector, the original
//        value will be used instead.

              if (count)
              {
                    *ptr_alpha = *(ptr_alpha-1);
                    *ptr_beta = *(ptr_beta-1);
              }
              ptr_Ao2++;
              ptr_alpha++;
              ptr_beta++;
              ptr_sample+=3;
          }
    }

// Set the last alpha value of error vector to gain factor since this
//        vector is corresponding to Ao2=10*Aomax which is the end point
//        of the linear interpolation.

    *ptr_alpha=gain_factor;

}   // End of member function.

//////////////////////////////////////////////////////////////////////////////
//
//  find_slope()
//
//  This member function is used to compute the slope of the alpha and beta
//  coefficients between two adjacent sampling points. This slope is
//  applied into the linear interpolation in order to generate the new
//  lookup table.
//

void Clinearlib::find_slope()
{

    for(count=0, ptr_Ao2=Ao2, ptr_alpha=alpha, ptr_beta=beta,
          ptr_alpha_slope=alpha_slope, ptr_beta_slope=beta_slope;
          count<SMP_SIZE1; count++)
    {
          dAo2=(*(ptr_Ao2+1))-(*(ptr_Ao2++));
          *(ptr_alpha_slope++)=((*(ptr_alpha+1))-(*(ptr_alpha++)))/dAo2;
          *(ptr_beta_slope++)=((*(ptr_beta+1))-(*(ptr_beta++)))/dAo2;
    }
}   // End of member function.

//////////////////////////////////////////////////////////////////////////////
//
//  compute_table()
//
//  This member function is used to compute a new predistorted lookup
//  table according to the accumulated table and the most recent
//  sampling data. The new predistorted lookup table is calculated
//  by using weighted sum. Therefore W_old+W_new must be equal to
//  one, and W_new/W_old is equal to the damming ratio. Besides
//  computing the new lookup table, this member function also
//  compute a new gain_factor factor according to the most updated
//  lookup table.
//

void Clinearlib::compute_table()
{

//  The following section is compute a new predistorted lookup table

    for(ptr_alpha=alpha, ptr_beta=beta, ptr_Ao2=Ao2,
          ptr_alpha_slope=alpha_slope, ptr_beta_slope=beta_slope,
          ptr_table=table, mag2=LUT_HLF_DMAG2, count=0,
          max_mag2=max_output_mag2=0.0; count < HLF_SIZE;
          count ++, mag2+=LUT_DMAG2)
    {
          if (mag2> *(ptr_Ao2+1))
          {
              ptr_Ao2++;
              ptr_alpha++;
              ptr_beta++;
              ptr_alpha_slope++;
              ptr_beta_slope++;
          }
          dmag2=mag2-(*ptr_Ao2);
          *(ptr_table++)=W_old*(*ptr_table)+
                W_new*((*ptr_alpha)+(*ptr_alpha_slope)*dmag2);

          *ptr_table=W_old*(*ptr_table)+
                W_new*((*ptr_beta)+(*ptr_beta_slope)*dmag2);
          output_mag2=(*(ptr_table-1)*(*(ptr_table-1))+
            *(ptr_table)*(*(ptr_table++)))*mag2;
          if (output_mag2>max_output_mag2)
          {
              max_output_mag2=output_mag2;
            max_mag2=mag2;
        }
    }

//  This section is used to compute the new gain control factor. This factor
//  is calculated according to the maximum magnitude of the F vector.
//  Here, the maximum output magnitude is assumed to be the
//  multiplication of the largest Ao and the last F vector. (i.e. since
//  the largest Ao in the lookup table is fixed, the last F vector
//  becomes the deterministic factor. If the magnitude of the last F
//  vector is larger than one, the difference will be used to adjust the
//  gain control factor.

    output_error_ratio=(max_output_mag2-ref_output_level)*
                           inv_ref_output_level;

    regulator_switch+=(output_error_ratio>0) ?
                          (regulator_switch<REG_SW_THSHLD) : 0;

    if ((regulator_switch>=REG_SW_THSHLD)&&
          (fabsl(output_error_ratio)>CLP_REG_TOLERANCE))
    {
            cout << "Output regulator in use" << endl;
            gain_factor*=(1-OUTPUT_DAMPING*output_error_ratio);
    }
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  update_table()
//
//  This member function is used to update the new calculated lookup table
//  into the C30. The array 'table' is divided into two parts. The
//  first and second parts store the new calculated alpha and beta
//  values respectively. The table is directly saved into the dual port
//  memory section. Once the update is completed, the 486 will signal
//  C30 to transfer the new lookup table from the dual port memory
//  section to the 0ws memory section by using C30's DMA controller.
//  In this case, C30 will not be interrupted during the transfer of
//  the new lookup table.
//

void Clinearlib::update_table()
{

// Wait for the dual port buffer is already.

    while(!dsptopc());

// if already, transfer the most update look-up table.
    WrBlkFlt(ALPHA_DUAL_ADR, BLK_SIZE, table);

// Acknowledge to C30 and enable the DMA controller to start to transfer
//  the new look-up table.

    pctodsp(Cinterface::ON);
    while(dsptopc());
    pctodsp(Cinterface::OFF);
}   // End of member function.

////////////////////////////////////////////////////////////////////////////////
//
//  update_table(const float &p_alpha, const float &p_beta)
//
//  This member function is similar to the update_table(). The only
//  difference is what the values will be transferred to dual port
//  buffer. In this function, 1 and 0 are transferred to the dual port
//  buffer as for alpha and beta values respectivly.
//

void Clinearlib::update_table(const float &p_alpha, const float &p_beta)
{

// Assume the characteristic of the amplifier

    //ldouble *max_a=table+HLF_SIZE-1,*max_b=table+BLK_SIZE-1;
    ulong temp_adr=ALPHA_DUAL_ADR;
    ldouble temp_amp=sqrtl(max_output_mag2/max_mag2);

    while(!dsptopc());
    for (ulong count=0;count<HLF_SIZE;count++)
    {
          PutFloat(temp_adr++,temp_amp);
          PutFloat(temp_adr++,0.0);
    }
    pctodsp(Cinterface::ON);
    while(dsptopc());
    pctodsp(Cinterface::OFF);
}   // End of member function.
```

List 13: 486 PC C++ program files for the enhanced linearizer.
File Name: Clineari.cpp.

```
#include <new.h>
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#include <math.h>
#include "Clineari.h"

//////////////////////////////////////////////////////////////////////////////
//
//  Clinarizer's constructor
//
//  Initialize the specific C30 board and check whether this board is
//  avaible for downloading the linearization program.
//
Clinearizer::Clinearizer(const ushort &board_no, const ulong &noloop)
              :Clinearlib(board_no)
{

    if (!Cinterface::board_status)
    {
          cerr << "\n\nError during board initialization !!!" << endl;
          state_reg=QUIT;
    }
    else
          if (!calibration())
          {
              cerr << "\n\nError during board calibration !!!" << endl;
              state_reg=QUIT;
          }
          else
          {
              set_for_linearization();
              counter=0;
              loopcounter=noloop;
              state_reg=START_LINEARIZATION;
          }
}   // End of Clinearizer's constructor.

//////////////////////////////////////////////////////////////////////////////
//
//  calibration()
//
//  Prompt the required procedure for the calibation and toggle the
//  calibration routine.
//
//  Return : 1 = Ok
//           0 = Fail
//
uchar Clinearizer::calibration()
{
    cout << "Please short the feedback loop on the board ("
         << Cinterface::board_ID << ") for calibration\n"
         << "Press any key to start the calibration routine"
         << endl;
    while(!kbhit());
    _getch();
    if (calibrating())
    {
          cout << "Please disconnect the short circuit around the feedback loop\n"
               << "Press any key to start the linearization routine"
               << endl;
          while(!kbhit());
          _getch();
          return 1;
    }
    return 0;
}   // End of member function.

//////////////////////////////////////////////////////////////////////////////
//
//  linearization()
//
//  This is a virtual member function. The function breaks down the
//  linearization algorithm into different stages.
//  (Note: The output signal is predistorted. No message is displayed
//           on the screen.
//
//  Return the current stage value.
//
uchar Clinearizer::linearization()
{

    switch (state_reg)
    {
          case START_LINEARIZATION:
              start();
              state_reg=GET_SAMPLES;
              return state_reg;
          case GET_SAMPLES:
              if(get_samples())
                    state_reg=GET_ENERGY;
              return state_reg;
          case GET_ENERGY:
              if(get_energy())
                    state_reg=DATA_CONVERSION;
              return state_reg;
          case DATA_CONVERSION:
              if (Clinearlib::regulator_switch<REG_SW_THSHLD)
                    power_regulator();
              data_coversion();
              state_reg=FIND_SLOPE;
              return state_reg;
          case FIND_SLOPE:
              find_slope();
              state_reg=COMPUTE_TABLE;
              return state_reg;
          case COMPUTE_TABLE:
              compute_table();
              state_reg=UPDATE_LUT;
              return state_reg;
          case UPDATE_LUT:
              update_table();
              state_reg=CHECK_ENDING;
              return state_reg;
          case CHECK_ENDING:
              counter++;
              state_reg=(counter==loopcounter) ? QUIT : GET_SAMPLES;
              return state_reg;
          case QUIT:
          default:
              state_reg=QUIT;
    }
    return state_reg;
}   // End of member function.

//////////////////////////////////////////////////////////////////////////////
//
//  setup_status()
//
//  Return the current board status.
//
uchar Clinearizer::setup_status()
{
    return Cinterface::board_status;
}   // End of member function.

//////////////////////////////////////////////////////////////////////////////
//
//  loop_counted()
//
//  Return the number of the loop the linearizer counted.
//
ulong Clinearizer::loop_counted()
{
    return counter;
}   // End of member function.

//////////////////////////////////////////////////////////////////////////////
//
//  CDUlinearizer's constructor
//
//  it is a empty constructor except for initializing the specific C30 board.
//
CDUlinearizer::CDUlinearizer(const ushort board_no, const ulong noloop)
                 :Clinearizer(board_no, noloop)
{}  // End of CDUlinearizer constructor.

//////////////////////////////////////////////////////////////////////////////
//
//  linearization()
//
//  This is a virtual member function from Clinearizer. The function breaks
//  down the linearization algorithm into different stages.
//  (Note: the output signal is uniform => no predistortion. Some messages
//           are displayed on the screen to indicate the current operating
//           condition.
//
//  Return the current stage value.
//
uchar CDUlinearizer::linearization()
{
    switch (state_reg)
    {
           case START_LINEARIZATION:
               start();
               state_reg=GET_SAMPLES;
               return state_reg;
           case GET_SAMPLES:
               cout << "\b->" << flush;
               if(get_samples())
```

```
                    state_reg=GET_ENERGY;
              return state_reg;
          case GET_ENERGY:
              if(get_energy())
                    state_reg=DATA_CONVERSION;
              return state_reg;
          case DATA_CONVERSION:
              if (Clinearlib::regulator_switch<REG_SW_THSHLD)
                    power_regulator();
              cout <<"\rU:"<< setw(4) << counter << " " << flush;
              data_coversion();
              state_reg=FIND_SLOPE;
              return state_reg;
          case FIND_SLOPE:
              find_slope();
              state_reg=COMPUTE_TABLE;
              return state_reg;
          case COMPUTE_TABLE:
              compute_table();
              state_reg=UPDATE_UT;
              return state_reg;
          case UPDATE_UT:
              update_table(1,0);
              state_reg=CHECK_ENDING;
              return state_reg;
          case CHECK_ENDING:
              counter++;
              state_reg=((counter==loopcounter) || _kbhit()) ?
                          QUIT : GET_SAMPLES;
              return state_reg;
          case QUIT:
          default:
              state_reg=QUIT;
    }
    return state_reg;
}   // End of member function.

/////////////////////////////////////////////////////////////////////////////
//
//  CDlinearizer's constructor
//
//  it is a empty constructor except for initializing the specific C30 board.
//
CDlinearizer::CDlinearizer(const ushort board_no, const ulong noloop)
               :Clinearizer(board_no, noloop)
{}  // End of CDlinearizer constructor.

/////////////////////////////////////////////////////////////////////////////
//
//  linearization()
//
//  This is a virtual member function from Clinearizer. The function breaks
//  down the linearization algorithm into different stages.
//  (Note: The output signal is predistorted. Some messages are displayed
//             on the screen to indicate the current operating condition.
//
//  Return the current stage value.
//
uchar CDlinearizer::linearization()
{

    switch (state_reg)
    {
          case START_LINEARIZATION:
              start();
              state_reg=GET_SAMPLES;
              return state_reg;
          case GET_SAMPLES:
              cout << "\b->" << flush;
              if(get_samples())
                    state_reg=GET_ENERGY;
              return state_reg;
          case GET_ENERGY:
              if(get_energy())
                    state_reg=DATA_CONVERSION;
              return state_reg;
          case DATA_CONVERSION:
              if (Clinearlib::regulator_switch<REG_SW_THSHLD)
                    power_regulator();
              cout << "\rP:"<< setw(4) << counter << " " << flush;
              data_coversion();
              state_reg=FIND_SLOPE;
              return state_reg;
          case FIND_SLOPE:
              find_slope();
              state_reg=COMPUTE_TABLE;
              return state_reg;
          case COMPUTE_TABLE:
              compute_table();
              state_reg=UPDATE_LUT;
              return state_reg;
          case UPDATE_LUT:
              update_table();
              state_reg=CHECK_ENDING;
              return state_reg;
          case CHECK_ENDING:
              counter++;
              state_reg=((counter==loopcounter) || _kbhit()) ?
                          QUIT : GET_SAMPLES;
              return state_reg;
          case QUIT:
          default:
              state_reg=QUIT;
    }
    return state_reg;
}   // End of member function.

//////////////////////////////////////////////////////////////////////////////
//
//  CNDUlinearizer's constructor
//
//  it is a empty constructor except for initializing the specific C30 board.
//
CNDUlinearizer::CNDUlinearizer(const ushort board_no, const ulong noloop)
                 :Clinearizer(board_no, noloop)
{}  // End of CDlinearizer constructor.

//////////////////////////////////////////////////////////////////////////////
//
//  linearization()
//
//  This is a virtual member function from Clinearizer. The function breaks
//  down the linearization algorithm into different stages.
//  (Note: the output signal is uniform => no predistortion.)
//
//  Return the current stage value.
//
uchar CNDUlinearizer::linearization()
{

    switch (state_reg)
    {
          case START_LINEARIZATION:
              start();
              state_reg=GET_SAMPLES;
              return state_reg;
          case GET_SAMPLES:
              if(get_samples())
                    state_reg=GET_ENERGY;
              return state_reg;
          case GET_ENERGY:
              if(get_energy())
                    state_reg=DATA_CONVERSION;
              return state_reg;
          case DATA_CONVERSION:
              if (Clinearlib::regulator_switch<REG_SW_THSHLD)
                    power_regulator();
              data_coversion();
              state_reg=FIND_SLOPE;
              return state_reg;
          case FIND_SLOPE:
              find_slope();
              state_reg=COMPUTE_TABLE;
              return state_reg;
          case COMPUTE_TABLE:
              compute_table();
              state_reg=UPDATE_UT;
              return state_reg;
          case UPDATE_UT:
              update_table(1,0);
              state_reg=CHECK_ENDING;
              return state_reg;
          case CHECK_ENDING:
              counter++;
              state_reg=(counter==loopcounter) ? QUIT : GET_SAMPLES;
              return state_reg;
          case QUIT:
          default:
              state_reg=QUIT;
    }
    return state_reg;
}   // End of member function.
```

List 14: 486 PC C++ program files for the enhanced linearizer.
File Name: Ctiming.cpp.

```
#include <iostream.h>
#include <iomanip.h>
#include <bios.h>
#include "Ctiming.h"

uchar Ctiming::timer(const unsigned control, const float duration,
                         const ulong numoftrial)
{

    float trial=(float)numoftrial;
    switch(control)
    {
          case SETTIME:
              _bios_timeofday( _TIME_GETCLOCK, &bstart );
              return 0;
          case SETSTART:
              _bios_timeofday( _TIME_GETCLOCK, &i_time );
              c_time = i_time;
              return 0;
          case SHOWTIME:
              _bios_timeofday( _TIME_GETCLOCK, &bend );
              if (numoftrial==0) trial=1;
              cout << "\n>>>>>> The TIME taken for one table transmission "
                     << setprecision(4)
                     << (((float)bend-(float)bstart)/trial/TICKPERSEC*1000)
                     << "ms\n" << endl;
              return 1;
          case CHECKTIME:
              _bios_timeofday( _TIME_GETCLOCK, &c_time );
              return ((((float)c_time-(float)i_time)/TICKPERSEC)>=duration);
          default:
              return 0;
          }
}
```

List 15: 486 PC C++ program files for the enhanced linearizer.
File Name: Ctest0.cpp.

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include "ctest0.h"

Ctest0::Ctest0(Clinearlib &testing):linearlib(testing)

{}

void Ctest0::condition()

{

    cout << "\n\nGain factor: " << setprecision(6)
            << (linearlib.gain_factor/ONE) << endl;
    cout << "Last sampling table address: " << hex
            << linearlib.last_smptbl << dec << endl;
    cout << "Next sampling table address: " << hex
            << linearlib.smptbl_st << dec << endl;

}

uchar Ctest0::last_table(const char *filename)

{
    ofstream data(filename);

    if (!data)
           return 1;
    else {

           ldouble mag;
           ushort    index_Ao2,index_alpha_table,index_beta_table;

           for(index_Ao2=index_alpha_table=0, index_beta_table=HLF_SIZE,
               mag=LUT_HLF_DMAG2; index_alpha_table<HLF_SIZE;
               index_alpha_table++, index_beta_table++, mag+=LUT_DMAG2) {

               index_Ao2 += (mag>linearlib.Ao2[index_Ao2+1]);

               data << setprecision(20)
                        << mag
                        << '\t'
                        << (linearlib.table[index_alpha_table]/ONE)
                        << '\t'
                        << (linearlib.table[index_beta_table]/ONE)
                        << '\t'
                        << (linearlib.alpha[index_Ao2]/ONE)
                        << '\t'
                        << (linearlib.beta[index_Ao2]/ONE) << endl;
               }
           return 0;
           }

}

uchar Ctest0::last_samples(const char *filename)

{

    ofstream data(filename);

    if (!data)
           return 1;
    else {

           for(ushort count=0; count<SMP_SIZE; count+=3)
               data << linearlib.sample_array[count].ulword
                        << '\t'
                        << linearlib.sample_array[count+1].of__.sword0
                        << '\t'
                        << linearlib.sample_array[count+1].of__.sword1
                        << '\t'
                        << linearlib.sample_array[count+2].of__.sword0
                        << '\t'
                        << linearlib.sample_array[count+2].of__.sword1 << endl;
           return 0;
           }

}

uchar Ctest0::adjusting_vector(const char *filename)

{

    ofstream data(filename);
    if (!data)
          return 1;
    else {

          for(ushort count=0; count<SMP_SIZE2; count++)
              data << setprecision(20)
                     << linearlib.Ao2[count]
                     << '\t'
                     << linearlib.alpha[count]
                     << '\t'
                     << linearlib.beta[count]
                     << endl;
          return 0;
          }

}
uchar Ctest0::show_slope(const char *filename)

{

    ofstream data(filename);

    if (!data)
          return 1;
    else {

          for(ushort count=0; count<SMP_SIZE1; count++)
              data << setprecision(20)
                     << linearlib.alpha_slope[count]
                     << '\t'
                     << linearlib.beta_slope[count] << endl;
          return 0;
          }
}
```

List 16: 486 PC C++ program files for the enhanced linearizer.
File Name: Ctest1.cpp.

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include "ctest1.h"

Ctest1::Ctest1(Clinearizer &testing):linearizer(testing)

{}

void Ctest1::condition()

{

    cout << "Gain factor: " << setprecision(6)
           << (linearizer.gain_factor/ONE) << endl;
    cout << "Last sampling table address: " << hex
           << linearizer.last_smptbl << dec << endl;
    cout << "Numbers of loop counted: " << linearizer.counter << endl;

}

uchar Ctest1::last_table(const char *filename)

{
    if (linearizer.counter!=0) {

          ofstream data(filename);
          if (!data) {
              cerr << "ERROR IN OUTPUT FILE !!!" << endl;
              return 1;
              }
          else {

              ldouble mag;
              ushort  index_Ao2,index_alpha_table,index_beta_table;

              for(index_Ao2=index_alpha_table=0, index_beta_table=HLF_SIZE,
                    mag=LUT_HLF_DMAG2; index_alpha_table<HLF_SIZE;
                    index_alpha_table++, index_beta_table++, mag+=LUT_DMAG2) {

                    index_Ao2 += (mag>linearizer.Ao2[index_Ao2+1]);

                    data << setprecision(20)
                         << mag
                         << '\t'
                         << (linearizer.table[index_alpha_table]/ONE)
                         << '\t'
                         << (linearizer.table[index_beta_table]/ONE)
                         << '\t'
                         << (linearizer.alpha[index_Ao2]/ONE)
                         << '\t'
                         << (linearizer.beta[index_Ao2]/ONE) << endl;
                    }
              return 0;
              }
          }
    else {
          cout << "\n No data has been sampled during the last operation !!!"
                << endl;
          return 2;
          }

}

uchar Ctest1::last_samples(const char *filename)

{
    if (linearizer.counter!=0) {

          ofstream data(filename);
          if (!data) {
              cerr << "ERROR IN OUTPUT FILE !!!" << endl;
              return 1;
              }
          else {

          for(ushort count=0; count<SMP_SIZE; count+=3)
              data << linearizer.sample_array[count].ulword
                     << '\t'
                     << linearizer.sample_array[count+1].of__.sword0
                     << '\t'
                     << linearizer.sample_array[count+1].of__.sword1
                     << '\t'
                     << linearizer.sample_array[count+2].of__.sword0
                     << '\t'
                     << linearizer.sample_array[count+2].of__.sword1 << endl;
              return 0;
```

```
                }
            }
    else {
            cout << "\n No data has been sampled during the last operation !!!"
                 << endl;
            return 2;
            }

}


void Ctest1::show_power()

{

    cout << linearizer.no_output_counted
           << '\t'
           << setprecision(20)
           << '\t'
           << linearizer.energy_I
           << '\t'
           << linearizer.energy_Q
           << '\t'
           << linearizer.output_power
           << '\t'
           << (linearizer.acc_power/linearizer.counter)
           << endl;

}
```

List 17: 486 PC C++ program files for the enhanced linearizer.
File Name: Utility.cpp.

```
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <math.h>
#include "utility.h"

void matrix_mod(double *I, double *Q, double *m, double *unknown,
                    double *matrix)
{

// the variable 'unknown' is a array stored a1,b1,a2,b2,fi

          double s1,s2,s3,s4;
          unsigned row,i;

// This routine is used to the 5x6 matrix which is based on the modulator
// impaiment equation and newton-raphson method. the first 5x5 matrix
// elements are represented the deviative of five functions wrt a1,b1,a2,
// b2 and fi. The last 5x1 matrix element are the negative value of these
// functions.
//
//  +-        -+ +-      -+   +-      -+
//  |  dfi    | |         |   |        |
//  |  ---    | | delta xi | = | - fi   |
//  |  dxi    | |         |   |        |
//  +-        -+ +-      -+   +-      -+
//     5x5                        5x1
//
// Since the simulation program from BOSS is very inefficient,
// another set of equations is dervied.
// These equations are shown on the Appendix.
//

          s3 = sin(unknown[4]);                                          // s3=sin(fi)
          s4 = cos(unknown[4]);                                          // s4=cos(fi)

          for(row=0,i=0;i<5;row+=6,i++)
          {
          // s1=a1*I+b1
                    s1 = *unknown * I[i] + unknown[1];
                    // s2=a2*Q+b2
                    s2 = unknown[2] * Q[i] + unknown[3];
                    // df/db1, df/da1 = I * df/db1
                    matrix[row] = I[i] * (matrix[row+1]=-2*(s1-s2*s3));
                    // df/db2, df/da2 = I * df/db2
                    matrix[row+2] = Q[i] * (matrix[row+3] = -2*(s2-s1*s3));
                    // df/dfi
                    matrix[row+4] = 2*s1*s2*s4;
                    // -f
                    matrix[row+5] = (s1*s4)*(s1*s4)+(s2-s1*s3)*(s2-s1*s3)-m[i]*m[i];
          }
}

void matrix_demod(double *I, double *Q, double *m, double *unknown,
                      double *matrix)
{

// the variable 'unknown' is a array stored a3,b3,a4,b4,fi

          double s1,s2,s3,s4,s5,s6;
          unsigned row,i;

// The function of this routine is similiar to the matrix_mod.
// The differnece is only on the impairment equations (i.e. for
// the demodulator instead of the modulator).

          s4 = 1/ cos(unknown[4]);                             // s4=1/cos(fi)
          s5 = tan(unknown[4]);                                          // s5=tan(fi)
          s6 = sin(unknown[4]);                                          // s6=sin(fi)

          for(row=0,i=0;i<5;row+=6,i++)
          {
                    // s1=(I-b3)/a3, s2=(Q-b4)/a4, s3
                    s3 = s4*((s1=(I[i]-unknown[1])/ *unknown)+
                              (s2=(Q[i]-unknown[3])/unknown[2])*s6);
                    // df/db3, df/da3 = s1 * df/db1
                    matrix[row] = s1 * (matrix[row+1]=2*s3*s4/ *unknown);
                    // df/db4, df/da4 = s2 * df/db4
                    matrix[row+2] = s2 * (matrix[row+3]=2*(s3*s5+s2)/unknown[2]);
                    // df/dfi
                    matrix[row+4] = -2*s3*(s3*s5+s2);
                    // -f
                    matrix[row+5] = s3*s3+s2*s2-m[i]*m[i];
          }
}

uchar gauss_jordan(double *coff, double *result)
```

```
{
    unsigned diagonal, row, col, row_max;
    unsigned row_size=NoOfUnknown-1,col_size=NoOfUnknown+1;
    double max,*matrix[NoOfUnknown],*row_tmp;

// Define the row pointers

    for(row=0;row<=row_size;row++)
          matrix[row]=&coff[row*col_size];

// This gauss-jordan method is modified so that only the upper triangle
// matrix is kept trace for the elimination. the lower triangle is just
// used for intermidiate calculate. In addition, each row of the matrixes
// is used a pointer to locate the actual position. Swapping two rows can
// be achieved by exchanging two rows's pointers only. Therefore, both
// techniques can improve the program performance.

// Since the privot value is located at the last column of the coff
// matrix, the searching of max privot value is not necessary.
// Therefore, there are only 4 searchings (i.e. number of unknown
// variables -1).

    for(diagonal=0;diagonal<row_size;diagonal++)
          {

// Find the max pivot value at the specific diagonal location

          max=matrix[diagonal][diagonal];
          row_max=diagonal;

          for(row=diagonal+1;row<=row_size;row++)
                    if (fabs(matrix[row][diagonal])>fabs(max))
                              {
                              max=matrix[row][diagonal];
                              row_max=row;
                              }

// if the initial diagonal value isn't the max value then
// swap two rows and noramlize the max row else normalize
// the current row

          if (max!=0.0)
                    {
                    if (row_max!=diagonal)

// swap two rows by exchanging two pointers

                              {
                              row_tmp=matrix[diagonal];
                              matrix[diagonal]=matrix[row_max];
                              matrix[row_max]=row_tmp;
                              }

// Normalize the entire row which has the max privot value.
// Note : only the upper triangle of the matrix has been normalized.

                    for(col=diagonal+1;col<col_size;col++)
                              matrix[diagonal][col]/=max;

// Perform the gauss-jordan elimination.
// Note : only the upper triangle of the matrix has been calculated

                    for(row=0;row<=row_size;row++)
                              if (row!=diagonal)
                                        for(col=diagonal+1;col<col_size;col++)
                                                  matrix[row][col]-=matrix[row][diagonal]*
                                                                        matrix[diagonal][col];
                    }
              else
                        break;
          }

// Since the elimination of the matrix has virtually ganuanteed that
// the max privot value is located in the last coff column, the
// procedure of seaching the max privot value is useless. In this
// case, a direct computation of gauss-jordan elimination is performed.
// In addition, the solution of this linear equation is placed into
// the result (5x1) matrix.

          if ((max!=0.0)&&(matrix[row_size][row_size]!=0.0))
              {
                    result[row_size]=matrix[row_size][NoOfUnknown]/
                                         matrix[row_size][row_size];
                    for(row=0;row<row_size;row++)
                        result[row]=matrix[row][NoOfUnknown]-
                                        matrix[row][row_size]*result[row_size];
                    return 1;
              }
          else return 0;
}

int invalid(size_t size)
```

```
{
    cerr << "Allocation failed, "<< size <<" bytes not available."<< endl;
    exit(1);
    return 0;
}

void retrieve_arguments(const int no_of_arg, char **const arg_value,
                              PROGARG *const progarg)

{

    for(int counter=0; counter<NO_BOARDS; counter++) {
          progarg[counter].noloop=NOLOOP;
          progarg[counter].run_type=RUNTYPE;
          }

    char *parameter,temp;
    uchar board_no;
    for (counter=1; counter<no_of_arg; counter++) {
          parameter=arg_value[counter];
          if (*(parameter++)=='/') {
              board_no=*(parameter++)-'0';
              if ((board_no>=0)&&(board_no<NO_BOARDS))
                    switch (*(parameter++)) {
                        case 'r':
                              temp = toupper(*parameter);
                              if ((temp=='U')||(temp=='P'))
                                  progarg[board_no].run_type=*parameter;
                              else
                                  cout << "ERROR in parameter /r, "
                                         << "use the default setting ("<< RUNTYPE
                                         << ")" << endl;
                              break;
                        case 'c':

                              if (!(progarg[board_no].noloop=(ulong)fabs(atof(parameter)))) {
                                  cout << "ERROR in parameter /c, "
                                         << "use the default setting "
                                         << NOLOOP << endl;
                                  progarg[board_no].noloop=NOLOOP;
                                  }
                              break;
                        case '?':
                              cout << "/r(x) : run type -> (u)nity or (p)redistorted" << endl
                                   << "/c(n) : (n)umber of loop counted" << endl;
                              exit(0);
                    }
              }
          }
}

uchar assign_objects(Clinearizer **const Linearizer, Ctestl **const Check,
            Ctiming **const timing, uchar *const Linearizer_status,
            const PROGARG *const prog_arg)

{
    uchar no_working_boards=0;
    ushort board_adr[]=BOARD_ADDRESS;

    for(uchar number=0; number<NO_BOARDS; number++) {
          switch(prog_arg[number].run_type) {

// Unit output to the load without any predistortion.

              case 'u':
                    Linearizer[number]=new CDUlinearizer(
                        board_adr[number], prog_arg[number].noloop);
                    Linearizer_status[number]=WORKING;
                    break;

// Predistort the modulated signal to the load.

              case 'p':
                    Linearizer[number]=new CDlinearizer(
                        board_adr[number], prog_arg[number].noloop);
                    Linearizer_status[number]=WORKING;
                    break;

// The following two cases are similar to the above two, but there is
//        no numberer and process status to be display.

              case 'U':
                    Linearizer[number]=new CNDUlinearizer(
                        board_adr[number], prog_arg[number].noloop);
                    Linearizer_status[number]=WORKING;
                    break;
              case 'P':
                    Linearizer[number]=new Clinearizer(
                        board_adr[number], prog_arg[number].noloop);
                    Linearizer_status[number]=WORKING;
                    break;
              default:
                    Linearizer_status[number]=NO_INIT;
```

```
                }

            if (Linearizer_status[number])
                if (Linearizer[number]->setup_status()) {
                        Check[number] = new Ctest1(*Linearizer[number]);
                        timing[number]=new Ctiming;
                        no_working_boards++;
                        }
                else {
                        delete Linearizer[number];
                        Linearizer_status[number]=NO_INIT;
                        }
            }
    return no_working_boards;
}
```

List 18: 486 PC C++ program files for the enhanced linearizer.
File Name: Run.cpp. <<Main Program>>.

```
#include <iostream.h>
#include <new.h>
#include "const.h"
#include "Ctest0.h"
#include "Ctest1.h"
#include "utility.h"
#include "Ctiming.h"
#include "Clineari.h"

void main(int argc, char *argv[])

{
    Clinearizer *Linearizer[NO_BOARDS];
    Ctest1          *Check[NO_BOARDS];
    Ctiming         *timing[NO_BOARDS];
    uchar Linearizer_status[NO_BOARDS];
    PROGARG         prog_arg[NO_BOARDS];

    retrieve_arguments(argc, argv, prog_arg);
    _set_new_handler(invalid);

    uchar no_working_boards=assign_objects(Linearizer, Check, timing,
                                                  Linearizer_status, prog_arg);

    if (no_working_boards) {

          cout << "Start linearization program." << endl;

          for(uchar number=0;number<NO_BOARDS; number++)
              if (Linearizer_status[number])
                    timing[number]->timer(Ctiming::SETTIME,0,0);

          for(number=0; no_working_boards; number=(number+1)%NO_BOARDS)
              if (Linearizer_status[number]==WORKING)
                    if (Linearizer[number]->linearization()==Clinearizer::QUIT) {
                        no_working_boards--;
                        Linearizer_status[number]=STOP;
                        }

          for(number=0; number<NO_BOARDS; number++)
              if (Linearizer_status[number])
                    timing[number]->timer(Ctiming::SHOWTIME,0,
                        Linearizer[number]->loop_counted());

          for(number=0; number<NO_BOARDS; number++)
              if (Linearizer_status[number]) {
                    Check[number]->condition();
                    Check[number]->last_samples();
                    Check[number]->last_table();
                    Check[number]->show_power();
                    delete Linearizer[number];
                    delete timing[number];
                    delete Check[ number];
                    }
    }
}
```

List 19: TMS320C30 DSP digitally emulated power amplifier program.

```
TxRxdata:
        PUSH    ST                          ; Save status reg. into the stack
        PUSH    idx_tbl                     ; Save idx_tbl reg. into the stack
        PUSH    tmpa_1                      ; Save tmpa_1 reg. into the stack

        LDI     *+adc_adr(adc_chA),rx_I     ; In Chan A,B
||      LDI     *+adc_adr(adc_chB),rx_Q     ; Note: There is a memory conflict.
        STI     mod_I,*+adc_adr(dac_chA)    ; Out Chan A,B
||      STI     mod_Q,*+adc_adr(dac_chB)    ; Note: There is a memory conflict.

        LDIU    *c26_adr,idx_tbl            ; load the new modulated I and Q signals
        STI     idx_tbl,@TEMP1

prdstn:
        LDIU    idx_tbl,mod_Q               ; Make a copy of the input modulated signals
        LSH     left16,idx_tbl              ; Convert the input modulated 16bits I channel signals
        ASH     right16,idx_tbl             ;      to 32bits
        ASH     right16,mod_Q               ; Convert the input modulated 16bits Q channel signals
                                            ;      to 32bits

* Putting the following command here is to eliminate one of pipeline conflict

        LDI     tbl_adr,tmpa_1              ; Load the loopup table address
        FLOAT   mod_Q,rx_Q                  ; Convert the 2's complement values to floating point values.
        MPYI    mod_Q,mod_Q                 ; Calculate the square of the Q signal

        FLOAT   idx_tbl,rx_I                ; Convert the 2's complement values to floating point values.
        MPYI    idx_tbl,idx_tbl             ; Calculate the square of the I signal
        ADDI    mod_Q,idx_tbl               ; Calculate the square magnitude of the signal

        LSH     tuncat,idx_tbl              ; Calculate the table index
        ADDI    hlfsize,tmpa_1              ; Calculate the look up address for beta

        MPYF    *+tbl_adr(idx_tbl),rx_I,mod_I     ; Compute Ip
        MPYF    *+tmpa_1(idx_tbl),rx_Q,mod_Q
        SUBF    mod_Q,mod_I
        MPYF    *+tbl_adr(idx_tbl),rx_Q,mod_Q     ; Compute Ip
        MPYF    *+tmpa_1(idx_tbl),rx_I
        ADDF    rx_I,mod_Q

        FIX     mod_I,rx_I                  ; Convert to fix point format
        FIX     mod_Q,rx_Q                  ; Convert to fix point format

amplifier:

        ASH     right16,rx_I
        ASH     right16,rx_Q

        MPYI    rx_I,rx_I
        MPYI    rx_Q,rx_Q
        ADDI    rx_I,rx_Q,tmpa_1

        LSH     tun_amp,tmpa_1
        LSH     1,tmpa_1
        ADDI    @AMP_DIS,tmpa_1

        MPYF    *tmpa_1,mod_I,rx_I          ; Compute Ip
        MPYF    *+tmpa_1(1),mod_Q,rx_Q
        SUBF    rx_Q,rx_I
        MPYF    *tmpa_1,mod_Q,rx_Q          ; Compute Ip
        MPYF    *+tmpa_1(1),mod_I,RR1
        ADDF    RR1,rx_Q

        FIX     mod_I
        FIX     mod_Q
        FIX     rx_I
        FIX     rx_Q

        STI     mod_I,@TEMP2
        STI     rx_I,@TEMP3

        LDI     one,tmpa_1                  ; Set tmpr_2 to one
        CMPI    @SMP_EN,zero_r              ; Check the sampling enable condition
        BNZD    engcal                      ; if not ready to sample then jump to predistion routine
        LDILE   zero,tmpa_1                 ; if SMP_EN >= 0 then without increase SMP_EN
        ADDI    @SMP_EN,tmpa_1
        STI     tmpa_1,@SMP_EN              ; Update SMP_EN value

        ANDN    upper,mod_Q                 ; Get the tx Q channel data and place into upper 16bits
        LSH     right16,mod_I               ; Get the tx I channel data and place into lower 16bits
        OR      mod_Q,mod_I                 ; Combine the tx data into one 32 bits data

        LDIU    @TEMP1,mod_Q

        STI     mod_Q,*dly_adr              ; QIc
||      STI     mod_I,*+dly_adr(1)          ;QIp

        LDIU    rx_I,mod_I
        ANDN    upper,rx_Q                  ; Get the rx Q channel data and place into upper 16bits
        LSH     right16,mod_I               ; Get the rx I channel data and place into lower 16bits
        OR      rx_Q,mod_I                  ; Combine the rx data into one 32 bits data

        STI     mod_I,*+dly_adr(2)

        STI     dpadr_r,@SMP_EN             ; if reach the bottom then disable sampling

engcal:
        LDI     @ENG_FLG,tmpr_2             ; Check whether the power computation routine is enable?
                                            ; If ENG_FLG=1, calculate the power
        BZD     quit_isr                    ; If ENG_FLG=0, not calculate the power

        ASH     right16,rx_I                ; Convert the sampled signals from 16bits integer to 32bits
        ASH     right16,rx_Q

        MPYI    rx_I,rx_I
        ADDI    @OUT_CNT,tmpr_2             ; Increment the symbol count variable by 1.
        STI     tmpr_2,@OUT_CNT             ; Save the updated symbol count variable.

        ADDI    @ACCE_I0,rx_I               ; Accummulate the I channel energy
        LDIU    @ACCE_I1,mod_I
        ADDC    zero_r,mod_I

        MPYI    rx_Q,rx_Q                   ; Calculate the Q channel energy
        ADDI    @ACCE_Q0,rx_Q               ; Accummulate the I channel energy
        LDIU    @ACCE_Q1,mod_Q
        ADDC    zero_r,mod_Q

        STI     rx_I,@ACCE_I0               ; Backup the accummulated energy for power computation
        STI     mod_I,@ACCE_I1              ;      in the external task.
        STI     rx_Q,@ACCE_Q0
        STI     mod_Q,@ACCE_Q1
*
* This section is used to compute the LUT index and predistort the modulated signals
*

quit_isr
        LDI     @TEMP2,mod_I
        LDI     @TEMP3,mod_Q
        POP     tmpa_1                      ; Recover tmpa_1 reg. value
        POP     idx_tbl                     ; Recover idx_tbl reg. value
        POP     ST                          ; Recover status reg. value
        RETI
        .end
```